SOFTWARE ENGINEERING

DDD Injecting Services on Entity Methods Calls

Asked 3 years, 8 months ago Active 6 months ago Viewed 10k times

13

2

Short format of question

Is it within best practices of DDD and OOP to inject services on entity method calls?

Long format example

Let's say we have the classic Order-LineItems case in DDD, where we have a Domain Entity called an Order, which also acts as the Aggregate Root, and that Entity is comprised not only of it's Value Objects, but also a collection of Line Item Entities.

Suppose we want fluent syntax in our application, so that we can do something like this (noting the syntax in line 2, where we call the `getLineItems` method):

```
$order = $orderService->getOrderByID($orderID);
foreach($order->getLineItems($orderService) as $lineItem) {
  ...
}
```

We don't want to inject any sort of LineItemRepository into the OrderEntity, as that is a violation of several principles I can think of. But, the fluency of the syntax is something that we really want, because it's easy to read and maintain, as well as test.

Consider the following code, noting the method `getLineItems` in `OrderEntity` :

```
interface IOrderService {
    public function getOrderByID($orderID) : OrderEntity;
    public function getLineItems(OrderEntity $orderEntity) : LineItemCollection;
}

class OrderService implements IOrderService {
    private $orderRepository;
    private $lineItemRepository;

    public function __construct(IOrderRepository $orderRepository, ILineItemRepository
$lineItemRepository) {
        $this->orderRepository = $orderRepository;
        $this->lineItemRepository = $lineItemRepository;
    }

    public function getOrderByID($orderID) : OrderEntity {
        return $this->orderRepository->getByID($orderID);
    }

    public function getLineItems(OrderEntity $orderEntity) : LineItemCollection {
        return $this->lineItemRepository->getLineItemsByOrderID($orderEntity->ID());
    }
}

class OrderEntity {
    private $ID;
    private $lineItems;

    public function getLineItems(IOrderServiceInternal $orderService) {
        if(!is_null($this->lineItems)) {
            $this->lineItems = $orderService->getLineItems($this);
        }
        return $this->lineItems;
    }
}
```

Is that the accepted way of implementing fluent syntax in Entities without violating core principles of DDD and OOP? To me it seems fine, as we're only exposing the service layer, not the infrastructure layer (that is nested within the service) object-oriented php domain-driven-design domain-model lazy-initialization

Share Improve this question Follow

asked Sep 25 '17 at 0:22

e_i_pi
759 1 6 17

6 Answers

Active Oldest Votes

12

It's totally fine to pass a Domain service in an entity call. Say, we need to calculate an invoice sum with some complicated algorithm that can depend on, say, a customer type. Here is what it might look like:

```
class Invoice
{
    private $currency;
    private $customerId;

    public function __construct()
    {
    }

    public function sum(InvoiceCalculator $calculator)
    {
        $sum =
            new SumRecord(
                $calculator->calculate($this)
            )
        ;

        if ($sum->isZero()) {
            $this->events->add(new ZeroSumCalculated());
        }

        return $sum;
    }
}
```

Another approach though is to separate out a business-logic that is located in domain service via domain events. Keep in mind that this approach implies just different application services, but the same database transaction scope.

The third approach is the one I'm in favor of: if I find myself using a domain service, that probably means I missed some domain concept, since I model my concepts primarily with nouns, not verbs. So, ideally, I don't need a domain service at all and a good part of all my business-logic resides in decorators.

Share Improve this answer Follow

answered Nov 2 '17 at 8:25
Vadim Samokhin
1,948 1 11 16

late to the party but...domain events are far from being a one-size-fits-all solution. If you end up handling those events in the same transaction scope of the event sender, you're just trying to decouple something you cannot decouple. domain events are good to trigger side-effects but in this example we are rather handling a "prerequisite", an information that should exist before our aggregate is loaded from the storage, so domain services are definetly the way :) – Carmine Ingaldi May 5 at 6:29

9

I am shocked to read some of the answers here.

It's perfectly valid to pass domain services into entity methods in DDD to delegate some business calculations. As an example, imagine your aggregate root (an entity) needs to access an external resource through http in order to do some business logic and raise an event. If you don't inject the service through the entity's business method, how else you would do it? Would you instantiate an http client inside your entity? That sounds like a terrible idea.

What's incorrect is to inject services in aggregates through its constructor, but through a business method it's ok and perfectly normal.

Share Improve this answer Follow

edited Nov 23 '20 at 11:44
Paulo Merson
141 6

answered Aug 23 '18 at 10:14
diegosasw
307 2 8

1 Why would the case you've given not be the responsibility of a Domain Service? – e_i_pi Aug 23 '18 at 23:59

1 it is a Domain Service, but it's injected in business method. The application layer is just an orchestrator, – diegosasw Sep 6 '18 at 9:10

I am not experienced in DDD but should not be Domain Service be called from Application Service and after Domain Service validation continue to call Entity methods via that Application Service ? I am facing the same issue in my project, because the Domain Service runs database call via repository... I don't know if this is okay. – Muflix May 20 '19 at 15:33

The domain service should orchestrate, if you call it from the application later it means you somehow process the response and then you do something with it. Maybe that sounds like business logic. If so, it belongs in the Domain layer and the application later simply resolves the dependency and injects it in the aggregate. The domain service could have injected a repository whose implementation hitting database should belong in infrastructure layer (just the implementation, not the interface/contract). If it describes your ubiquitous language, it belongs in domain. – diegosasw May 20 '19 at 19:39

External resources are to be accessed from within Application layer, not Domain, which is encapsulated in itself. It is Application that orchestrates both domain objects and interaction with external systems. Still no issue injecting a Domain service into an Entity. – PavelS Jul 17 '20 at 15:30

8

> Is it within best practices of DDD and OOP to inject services on entity method calls?

No, you should not inject anything inside your domain layer (this includes entities, value objects, factories and domain services). This layer should be agnostic of any framework, 3rd party libraries or technology and should not make any IO calls.

```
$order->getLineItems($orderService)
```

This is wrong as the Aggregate should not need anything else but itself to return the order items. The *entire* Aggregate should be already loaded before its method call. If you feel that this should be lazy loaded then are two possibilities:

1. Your Aggregates boundaries are wrong, they are too large.
2. In this usecase you use the Aggregate only for reading. The best solution is to split the write model from the read model (i.e. use CQRS). In this *cleaner* architecture you are not allowed to query the Aggregate but a read model.

Share Improve this answer Follow

answered Sep 25 '17 at 5:10
Constantin Galbenu
3,044 7 14

If i need database call for validation, i have to call it in application service and pass an result to the domain service or directly into aggregate root rather then inject repository into domain service ? – Muflix May 20 '19 at 15:37

1 @Muflix yes, that's right – Constantin Galbenu May 20 '19 at 16:32

But you can inject a domain service into a domain entity, can't you? – PavelS Jul 17 '20 at 15:44

1 @MarkusKnappenJohansson class or interface, it doesn't matter. The Aggregate should not call external services (effectively). – Constantin Galbenu Sep 28 '20 at 10:27

1 While it might be a very good practice not to inject and I can agree that it should be avoided. In the case with a unique email or username-invariant, I can only see two solutions to this, either inject a service/interface that can validate or fire a domain event before setting the value (the latter being more complicated). Otherwise if a domain service was responsible for validation and there was a setter on the Aggregate, nothing would prevent a developer from setting the value before validating it... or How would you solve this? – Markus Knappen Johansson Oct 4 '20 at 13:17

6

The key idea in DDD tactical patterns: the application accesses all data in the application by acting on an aggregate root. This implies that the only *entities* which are accessible outside of the domain model are the aggregate roots.

The Order aggregate root would never yield a reference to its lineitem collection that would allow you to modify the collection, nor would it yield a collection of references to any line item that would allow you to modify it. If you want to change the Order aggregate, the hollywood principle applies: "Tell, don't ask".

Returning *values* from within the aggregate is fine, because values are inherently immutable; you can't change my data by changing your copy of it.

Using a domain service as an argument, to assist the aggregate in providing the correct values, is a perfectly reasonable thing to do.

You wouldn't normally use a domain service to provide access to data that is inside the aggregate, because the aggregate should already have access to it.

```
$order = $orderService->getOrderByID($orderID);
foreach($order->getLineItems($orderService) as $lineItem) {
  ...
}
```

So that spelling is weird, if we are trying to access this order's collection of line item values. The more natural spelling would be

```
$order = $orderService->getOrderByID($orderID);
foreach($order->getLineItems() as $lineItem) {
  ...
}
```

Of course, this pre-supposes that the line items have been loaded already.

The usual pattern is that the load of the aggregate will include all of the state required for the particular use case. In other words, you may have several *different* ways of loading the same aggregate; your repository methods are fit for purpose.

This approach isn't something that you will find in the original Evans, where he assumed that an aggregate would have a single data model associated with it. It falls more naturally out of CQRS.

Share Improve this answer Follow

answered Sep 25 '17 at 3:28
VoiceOfUnreason
26.7k 1 34 62

Thanks for this. I've now read about half of the "red book", and had my first taste of properly applying the Hollywood Principle in the infrastructure layer. Re-reading all of these answers, they all make good points, but I think yours has some very important points regarding scope of `lineItems()` and preloading upon first retrieval of the Aggregate Root. – e_i_pi Nov 2 '17 at 22:44

4

The answer is: definitely NO, avoid passing services in entity methods.

The solution is simple: just let the Order repository return the Order with all of its LineItems. In your case the aggregate is Order + LineItems, so if the repository does not return a complete aggregate, then it's not doing its job.

The broader principle is: keep functional bits (e.g. domain logic) separate from non-functional bits (e.g., persistence).

One more thing: if you can, try to avoid doing this:

```
$order = $orderService->getOrderByID($orderID);
foreach($order->getLineItems() as $lineItem) {
  ...
}
```

Do this instead

```
$order = $orderService->getOrderByID($orderID);
$order->doSomethingSignificant();
```

In object-oriented design, we try to avoid fishing around in an object data. We prefer to ask the object to do what we want.

Share Improve this answer Follow

answered Sep 25 '17 at 5:37
xpmatteo
351 2 6

3

Generally speaking value objects belonging to aggregate don't have repository by themselves. It's aggregate root's responsibility to populate them. In your case, it's your OrderRepository's responsibility to both populate Order entity and OrderLine values objects.

Regarding, the infrastructure implementation of the OrderRepository, in case ORM, it's one-to-many relationship, and you can elect either eagerly or lazy load the OrderLine.

I'm not sure what your services exactly mean. It's quite close to "Application Service". If this is the case, it's generally not a good idea to inject the services to Aggregate root/Entity/Value Object. Application Service should be the client of Aggregate root/Entity/Value Object and Domain Service. Another thing about your services is, exposing value objects in Application Service is not a good idea either. They should be accessed by aggregate root.

Share Improve this answer Follow

answered Sep 25 '17 at 3:32
ivenxu
582 2 6