Should service layer accept a DTO or a custom request object from the controller?

Asked 1 year, 2 months ago Active 1 year, 2 months ago Viewed 2k times

8

As the title suggests what is the best practice when designing service layers?. I do understand service layer should always return a DTO so that domain (entity) objects are preserved within the service layer. But what should be the input for the service layer from the controllers?

I put forward three of my own suggestions below:

Method 1: In this method the domain object (Item) is preserved within the service layer.

```
class Controller
{
    @Autowired
    private ItemService service;

    public ItemDTO createItem(IntemDTO dto)
    {
        // service layer returns a DTO object and accepts a DTO object
        return service.createItem(dto);
    }
}
```

Method 2: This is where the service layer receives a custom request object. I have seen this pattern extensively in AWS Java SDK and also Google Cloud Java API

```
class Controller
{
    @Autowired
    private ItemService service;

    public ItemDTO createItem(CreateItemRequest request)
    {
        // service layer returns a DTO object and accepts a custom request object
        return service.createItem(request);
```

Method 3: Service layer accepts a DTO and returns a domain object. I am not a fan of this method. But its been used extensively used at my workplace.

```
class Controller
{
    @Autowired
    private ItemService service;

    public ItemDTO createItem(CreateItemRequest request)
    {
        // service layer returns a DTO object and accepts a DTO object
        Item item = service.createItem(request);
        return ItemDTO.fromEntity(item);
    }
}
```

If all 3 of the above methods are incorrect or not the best way to do it, please advise me on the best practice.

rest dto service-layer

edited Dec 30 '18 at 8:48

asked Dec 29 '18 at 3:22

Vino
1,203 1 10 22

I'm not sure if this question has anything to do with `domain-driven-design` . Could you elaborate how DDD comes into picture here. – Ankit Vijay Dec 29 '18 at 21:50

Sorry. I have removed DDD from the tags list – Vino Dec 30 '18 at 8:48

2 Answers

4

I'm from `C#` background but the concept remains same here.

In a situation like this, where we have to pass the parameters/state from application layer to service layer and, then return result from service layer, I would tend to follow separation-of-concerns. The service layer does not need to know about the `Request` parameter of you application layer/ controller. Similarly, what you return from service layer should not be coupled with what you return from your

For the above example, I would do something like this:

```
class Controller
{
     @Autowired
     private ItemService service;

     public ItemResponse createItem(CreateItemRequest request)
     {
        var creatItemDto = GetDTo(request);
        var itemDto = service.createItem(createItemDto);
        return GetItemResponse(itemDto);
    }
}
```

This may feel like more work since now you need to write addional code to convert the different objects. However, this gives you a great flexiblity and makes the code much easier to maintain. For example: `CreateItemDto` may have additional/ computational fields as compared to `CreateItemRequest` . In such cases, you do not need to expose those fields in your `Request` object. You only expose your `Data Contract` to the client and nothing more. Similarly, you only return the relevant fields to the client as against what you return from service layer.

If you want to avoid manual mapping between `Dto` and `Request objects` C# has libaries like `AutoMapper` . In java world, I'm sure there should be an equivalent. May be ModelMapper can help.

edited Jan 1 '19 at 21:54

answered Dec 30 '18 at 21:59

Ankit Vijay

1,957 20 33

I think you may have missed it by mistake. `createItemDto` must be the input to `service.createItem()` as opposed to `request` – Vino Jan 1 '19 at 11:47

@Vino, it was a typo. Have fixed it. Thanks! – Ankit Vijay Jan 1 '19 at 21:54

+1. In typical backends that serve as APIs to frontends, the response is made up of two parts: the HTTP status code and response headers if any, and the value passed in the form of JSON or XML formats. The output data is derived with the help of a `Serializer` , meant for that specific service. – Subhash Bhushan May 31 '19 at 3:14

4

Conceptually speaking, you want to be able to reuse the service/application layer across presentation layers and through different access ports (e.g. console app talking to your app through a web socket). Furthermore, you do not want every single domain change to bubble up into the layers above the application layer.

The controller conceptually belongs to the presentation layer. Therefore, you wouldn't want the application layer to be coupled upon a contract defined in the same conceptual layer the controller is defined in. You also wouldn't want the controller to depend upon the domain or it may have to change when the domain changes.

You want a solution where the application layer method contracts (parameters & return type) are expressed in any Java native types or types defined in the service layer boundary.

If we take an IDDD sample from Vaughn Vernon, we can see that his application service method contracts are defined in Java native types. His application service command methods also do not yield any result given he used CQRS, but we can see query methods do return a DTO defined in the application/service layer package.

> In the above listed 3 methods which ones are correct/wrong?

Both, #1 and #2 are very similar and could be right from a dependency standpoint, as long as `ItemDto` and `CreateItemRequest` are defined in the application layer package, but I would favor #2 since the input data type is named against the use case rather than simply the kind of entity it deals with: entity-naming-focus would fit better with CRUD and because of that you might find it difficult to find good names for input data types of other use case methods operating on the same kind of entity. #2 also have been popularized through CQRS (where commands are usually sent to a command bus), but is not exclusive to CQRS. Vaughn Vernon also uses this approach in the IDDD samples. Please note that what you call **request** is usually called **command**.

However, #3 would not be ideal given it couples the controller (presentation layer) with the domain.

> For example, some methods receive 4 or 5 args. According to Eric Evans in Clean Code, such methods must be avoided.

That's a good guideline to follow and I'm not saying the samples couldn't be improved, but keep in mind that in DDD, the focus is put on naming things according to the Ubiquitous Language (UL) and following it as closely as possible. Therefore, forcing new concepts into the design just for the sake of grouping arguments together could potentially be harmful. Ironically, the process of attempting to do so may still offer some good insights and allow to discover overlooked & useful domain concepts that could enrich the UL.

PS: Robert C. Martin has written Clean Code, not Eric Evans which is famous for the blue book

plalx
36k 4 56 76

So I am having issues understanding your answer. In the above listed 3 methods which ones are correct/wrong?. Also I looked at code samples you linked. I am not entirely sure of some of the practices used in them. For example, some methods receive 4 or 5 args. According to Eric Evans in Clean Code, such methods must be avoided. – Vino Dec 30 '18 at 8:51

@Vino FYI - I updated the answer since you posted your comment. – plalx Jan 15 '19 at 18:54

Thanks for your insight. Yes, I mixed up the author's name. – Vino Jan 15 '19 at 23:48

@Vino Well, does my answer covers your question? Do you have any more questions? – plalx Jan 16 '19 at 0:48