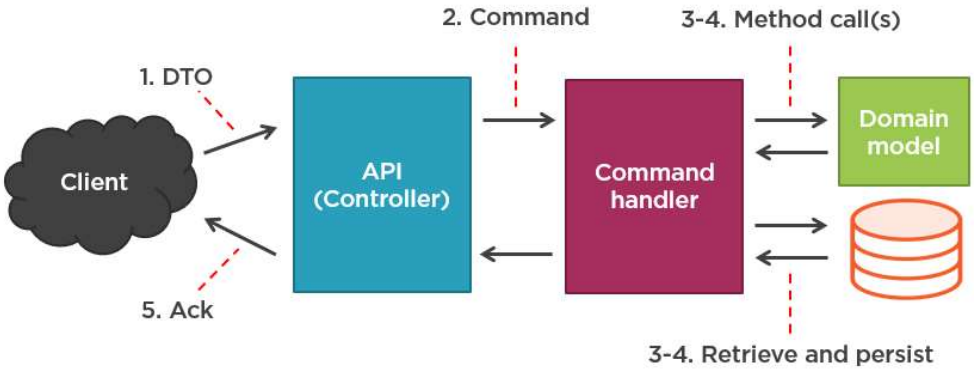
When to validate commands in CQRS?

February 20, 2019

I'm continuing the series about CQRS to supplement my recent CQRS in Practice course. There's a question that I didn't cover in the course and that was raised at least twice since then: when to validate commands in CQRS?

CQRS commands, command handlers, and validation

In a typical CQRS application with a single database, the write side looks like this:



CQRS write side

- The client sends a request to the API (the server). The request contains data that is represented by a DTO (data contract).
- The server routes the request to a controller which then transforms the incoming DTO into a command and dispatches it to a command handler.
- The command handler is where the application logic resides. It retrieves necessary data from the database, delegates decision-making to domain classes, and persists the results of those decisions back to the database.
- Finally, the client receives an acknowledgment: an indication of either success or failure.

So, where in this picture does validation lies? When exactly should you validate a command before acting upon it?

When it comes to preliminary validation, such as checking for non-nullability and such, you could put precondition checks into commands themselves:

Vladimir Khorikov

```
public sealed class EnrollCommand : ICommand
{
    public long StudentId { get; }
    public string Course { get; }
    public string Grade { get; }

    public EnrollCommand(long studentId, string course, string grade)
    {
        if (course == null || grade == null) // Precondition checks
            throw ArgumentException();

        Id = id;
        Course = course;
        Grade = grade;
    }
}
```

This could help avoid transmitting invalid commands to command handlers and follow the fail fast principle. You can even introduce a static factory method returning a Result instance, similar to Value Objects:

```
public sealed class EnrollCommand : ICommand
{
    public long StudentId { get; }
    public string Course { get; }
    public string Grade { get; }

    /* ... */

    public static Result<EnrollCommand> Create(long studentId, string course, string
grade)
    {
        /* Validate, return either success or failure */
    }
}
```

Would it be a good idea?

To answer this question, we need to revisit what a command is. A command is a message that tells your application to do something. **The application can either accept or reject this message**, depending on the application's current state. There's no guarantee that the application will go ahead and execute the command.

This is one of the differentiating factors between commands and events. Unlike a command, a domain event represents a fact that already happened and the application can't do anything about it. It can either take it into account or ignore, but not change or reject it.

Because a command represents things the client asks of your application, that could be anything, including something invalid. Therefore, **commands shouldn't have any invariants attached to them**. The application, in turn, is free to reject an invalid command. It means that a command should be a plain bag of properties with no validations:

– Entity vs Value Object: the ultimate list of differences
– DTO vs Value Object vs POCO
– 3 misuses of ?. operator in C# 6
– Specification pattern: C# implementation
– Database versioning best practices
– Unit testing private methods
– Functional C#: Handling failures, input errors
– REST API response codes: 400 vs 500

RECENT ARTICLES

– Database and Always-Valid Domain Model
– Specification Pattern vs Always-Valid Domain Model
– Nulls in Value Objects
– Validation and DDD: New online course
– Static methods considered evil?
– Always-Valid Domain Model
– How to handle unique constraint violations
– C# 9 Records as DDD Value Objects
– Domain model purity and lazy loading
– Domain model purity and the current time

» All articles

```
public sealed class EnrollCommand : ICommand
{
    public long StudentId { get; }
    public string Course { get; }
    public string Grade { get; }

    public EnrollCommand(long studentId, string course, string grade)
    {
        // No invariant checks
        Id = id;
        Course = course;
        Grade = grade;
    }
}
```

So where to do the validation then? There are a couple of options: either before dispatching the command (in the controller) or after (in the command handler).

Ideally, all validations should be done after the command dispatch. You want to keep the controller as thin as possible - it should only be responsible for the built-in ASP.NET functionality (such as routing) and converting DTOs into commands. Everything else should go to the domain model or commands handlers. This will help you keep a good separation between ASP.NET-related and application-related concerns.

Even trivial validations, such as checking for nulls, field lengths and so on, should go to the domain model in an ideal case. That's because the information regarding what constitutes a valid email, course name, or student address is part of the domain knowledge and should be kept in the core domain layer (preferably in value objects).

The drawback of this approach is that it's quite daunting, especially in projects without much complexity. In such simple projects, it may be an overkill to create value objects for each concept in the domain model, such as Email, CourseName and so on. (Check out this article to decide when to create a value objects: Value Objects: when to create one?)

The simpler approach is to rely on the built-in validation mechanism, such as ASP.NET attributes, or tools like FluentValidator. In such a scenario, you will validate commands before they get to command handlers. Which is not ideal from a pureness perspective, but is fine overall if you think that the simplicity gain is worth it. I wrote about a similar topic in this article: Validation and DDD .

Note two things:

- Even if you take the simpler approach (attributes over value objects), complex validations (such as checking for student's email uniqueness) should still be done in the command handlers.
- And if you have an HTML client, you will still need to duplicate the validations on the UI, at least the simple ones, for better user experience. No one wants to wait for a server roundtrip just to see that the email they entered is invalid.

Summary

- Commands should have no invariants attached to them. The server is free to reject an invalid command.
- Ideally, all validations should reside either in command handlers or in the domain model. The drawback of this approach is its complexity.
- In simpler projects, use attributes to streamline the validation. This approach is not as pure, but it's simpler.

Other articles in the series

- Are CQRS commands part of the domain model?
- **When to validate commands in CQRS? (this post)**
- CQRS and exception handling

← Are CQRS commands part of the domain model?

CQRS and exception handling →

Subscribe

I don't post everything on my blog. Don't miss smaller tips and updates. Sign up to my mailing list below.

Your email Sign up

Comments

Kamil Grzybek • 3 years ago

Vladimir, great article again!

What a coincidence that I wrote about similar topic yesterday. I proposed to validating commands using Pipeline Pattern - do validation **before** Command Handler execution but still in Application Services layer using Behaviors. What do you think about this solution? For me it is the "less trade-off" solution. http://www.kamilgrzybek.com...

1 • Reply • Share ›

Vladimir Khorikov Mod ➚ Kamil Grzybek • 3 years ago

Good approach. It ties back to my remark about the use of FluentValidator.

I disagree with the segregation of different types of validation, though. Data validation _is_ business rule validation. The only reason why they are often segregated is to justify having two places to handle validation (e.g. FluentValidation and command handlers). This is often fine but a proper assessment is to say that it's a trade-off: in this particular project, the convenience of FluentValidator is more preferable than pureness.

• Reply • Share ›

Kamil Grzybek ➚ Vladimir Khorikov • 3 years ago

I understand your point of view but I like this segregation, for me it is easier to understand and reason about system design. Sometimes the boundary between these types of validation is blurry. For example sometimes maximum length of property is system requirement (because you don't want to store all strings as varchar(max) types in db), sometimes it is business rule (for example Driving License number has to have exact number of characters). After my article publication I have recently come across an Daniel's Whittaker article http://danielwhittaker.me/2.... He wrote about similar segregation (superficial validation vs domain validation). The naming is different (maybe his is better, I don't know) but approach is pretty the same.

1 • Reply • Share ›

alireza Rahmani khalili • 4 months ago

it is super stranger that you do not mention command is in application layer

• Reply • Share ›

Yazan Aboudi • a year ago • edited

Thanks for the article and your thoughts. This is a topic I am currently struggling with. I have a few thoughts that I was hoping to get your opinion on. Humour me a little.

It seems like you are implying that a command is not part of the application domain. Why is that? Enrolling a student (your example) should be an operation that is recognized by domain experts. I

would argue that it makes sense to consider it as part of your domain model. Because of this, I thought that maybe command can be bag of Value Objects. The command will only try to instantiate those value objects inside of it. So technically, the error is still raised by the domain layer as it is thrown upon construction of Value Objects in the Command. This way, your command can be valid by the mere fact that it has been instantiated.

Doing so would not address the fact that there are operational invariants that should be checked. That part could then be done on the domain service / domain layer.

I kinda feel like this suggestion is highly cohesive and very much abides by Single Responsibility Principle. The validations thrown by Commands are due to its inability to instantiate value objects and the errors thrown past the dispatch is due to inability to conduct business transaction.

What do you think of this suggestion of considering a Command to be part of the domain model?

^ | v • Reply • Share ›

Vladimir Khorikov Mod ➚ Yazan Aboudi • 10 months ago

Commands are part of the domain layer (https://enterprisecraftsman...). Regarding whether they can contain VOs -- they actually can. Check out this piece: https://khorikov.org/posts/... . Regarding where to do validation, I recommend this approach: https://enterprisecraftsman...

This statement in the article is probably not entirely correct : "Commands should have no invariants attached to them". What I meant by that is that you can't throw exceptions when checking those invariants and must do all the validation prior instantiating a command. This topic is covered the last article I referenced.

^ | v • Reply • Share ›

Yazan Aboudi ➚ Vladimir Khorikov • 10 months ago

Thank you so much for replying and clarifying. I see what you mean now. If I can reiterate the last article you linked, you are suggesting that we delegate all request validations to VOs. If there is an error, it should be raised. If not, then proceed to creating your Command object. Did I get that right? That is a better way to go about for sure. It has the added benefit of DI/IoC in the context of command creation. Thank you Vladimir

^ | v • Reply • Share ›

Vladimir Khorikov Mod ➚ Yazan Aboudi • 10 months ago

Yes, that's correct. Alternatively, you can do the same in the controller prior to instantiating the command, but that's more tedious.

^ | v • Reply • Share ›

Tony B • a year ago • edited

I'm working on a .NET Core POC using your DDD+CQRS recipes, and I'm trying to introduce OpenAPI/Swagger for the API documentation. Normally swagger encourages you to add the DataAnnotations in the Dtos for validation. Vlad - I'm curious what you recommend for OpenAPI, or API documentation in general? Is there a recipe that you have found works for you?

^ | v • Reply • Share ›

Vladimir Khorikov Mod ➚ Tony B • a year ago

In my current project, we've defined custom Swagger attributes and use them as annotations in DTOs. They refer to value objects for validation rules. For example, the regex for email validation is defined in EmailAddress value object as a public constant, and the attribute looks like this:
[SwaggerRegex(EmailAddress.Regex)]

^ | v • Reply • Share ›

Roger Zanelato • a year ago

Thanks, great article!!

For what I have read in others comments below and also other posts, for a complete validation scenary we'd have to do validation in two locals:
- CommandHandler (check if the necessary resources and relationships necessary exists in DB for example)
- Aggregates (business validation)

It's probably a silly question, but if the Command needs that the resource exists to be valid, we couldn't say the same for the Aggregate? And if so, all this validation shouldn't be together in some way?

^ | v • Reply • Share ›

nemke • 2 years ago

If I want to check for command idempotence (e.g. to check if some resource is already created in

db), should I put this validation in command handler?

Reply • Share ›

Vladimir Khorikov Mod ➚ nemke • 2 years ago

Yes

1 Reply • Share ›

Alexander Langer • 3 years ago • edited

Nice article. However, I tend to disagree a bit on the validation. :-)

For domain validation (like validity of an email address), I completely agree that this should NOT be validated within the command itself. In commands, an email property should be a simple string, and will only translated into an Email value object in the domain layer/command handler. So, any "domain invariants" should indeed be checked in the domain layer and not within the command (or client etc).

However, commands are messages, and as such a contract between the caller and the callee. Therefore, you may very well define in the contract that, say, arguments/properties must not be null or empty strings etc. This are trivial checks that are even supported programmatically (say, by attributes) by many third party tools. C# will soon even incorporate non-nullable references as a language feature.

If it's part of the contract, in my opinion the contract should be enforced as early as possible - which is the creation time of the object (command).

YMMV however on whether non-nullable fields or non-empty strings should be part of a contract. If you exchange messages with other systems that are not directly using the C#-classes but do, say, send JSON messages that are mapped to C#-classes, then you'd better check those in the command handler. If both, the callee and the caller are within the same system and use the *same* message classes, I prefer to enforce these invariants in the contract between callee and caller. Also helps a lot with unit testing.

Reply • Share ›

David Keaveny ➚ Alexander Langer • 3 years ago

It's definitely a case of "why not both?"; if we're talking C# then I use data annotations on the properties of the command object, to identify required fields, or set min/max string lengths/numeric ranges. That all gets picked up by a ModelValidatingCommandHandler decorator so it gets applied consistently across my commands. The other big plus here is that tools that generate API specifications (e.g. for Swagger/OpenAPI) will use those data annotations to mark up the specifications for API consumers.

Then within each CommandHandler implementation, I will have validation of business rules (no changing historical pay rates on a cost centre that has been used in a payroll - not the sort of thing that can be expressed as a data annotation!); those validation errors will get returned using the same data structure as the data annotation failures, so that the client doesn't have to handle the two different types of validations, well, differently.

1 Reply • Share ›

Vladimir Khorikov Mod ➚ Alexander Langer • 3 years ago

Commands shouldn't be raised by the application itself, all commands are coming from the outside of the app. You can't really enforce a contract in such circumstance, all you can do is reject the command with a 400/4xx response.

You are right that contract enforcement is a preferable behavior for classes communicating inside the same system. With commands, it's always external systems calling your system, though, so fail fast doesn't apply.

Reply • Share ›

Alexander Langer ➚ Vladimir Khorikov • 3 years ago

Hmm, in your figure above, clients communicate with your system through an API (Controller), not through commands. External clients do not even know about those commands, they only use DTOs (as depicted in the figure).

The contract in question is only between the API controller and the command handlers, which is not "from the outside of the app". The API controller is an

Top