Set Based Consistency Validation

By Yvonne Ceelie

November 11 2020

Blog by AxonIQ

6 min

Requests like “How do I ensure the email address in a User aggregate is unique?” or “How do I ensure usernames are only used once on command handling?” reach us quite often. We can rephrase them: *How do I consistently validate consistency among a set in such a system?* None of these are Axon-specific problems, really, but more so issues you would encounter when using CQRS in combination with Event Sourcing. Due to this, they crop up when using Axon Framework. Before diving into how to tackle this with Axon, let us first explain the problem we face here.

Why is this a problem?

So, why is set-based consistency validation an issue when you have an application based on CQRS and Event Sourcing? The main concern here is that CQRS imposes a separation of your model into two distinct components: the command and the query model. The command model is intended to perform the business validation in the application. In contrast, the query model is mainly used to respond to any queries the application can handle. The shift in prioritization allows both models to be optimized for their intended uses. The query model will typically contain the query responses as efficiently as possible, or how they will be used. The main focus of the command model, on the other hand, is to maintain the consistency boundary of the application.

When drafting up the command model, this will mostly be done in the form of aggregates. This concept originates from DDD, describing itself as “*a group of associated entities regarding data changes act as a single unit*”. Additionally, “*external references are* *restricted to a single member of the aggregate, designated as the root”*. Furthermore, “*a set of consistency rules apply within the aggregate’s boundaries”*. To keep an aggregate maintainable and compliant to this set of rules, the group of entities typically never span the entirety of a system. Instead, smaller manageable groups of entities are used. If it encompassed a large chunk of the entire system, there would be no way to guarantee the consistency boundary and be performant at the same time since the entire aggregate should be locked for every operation.

It is with this we can conceive why set-based consistency validation becomes even more complex. The aggregate typically encompasses a small portion of the model, whereas a set can be any size. Therefore, an aggregate can only guard its own consistency boundary, disregarding the entirety of the set. So, we need a different *thing* to allow for set-based validation. A logical first assumption would be to use a query model from our application to solve our problem. However, as CQRS dictates, the synchronization between both models takes some time and introduces *eventual consistency*. Mainly if events are used to drive both models, we will be held back by this. The query model cannot guarantee it knows the entire set due to this, and as such, it cannot be our solution to the problem.

So, how do we resolve the question: *How can I check if an account with a particular email address already exists when doing CQRS*? Well, we should look further into our command model. It does not necessarily exist solely from aggregates. A command model can contain *any* form of the data model. Thus, to solve the set-based consistency validation for email addresses, we can introduce a small look-up table containing all the addresses. It is this model which can be consulted during this process to allow for this form of validation. To ensure this other model deals with the problem correctly, any changes occurring on the model should be immediately consistent with the rest. In Axon, there are several ways to make this model consistent and to use it for validation. The rest of this blog will focus on updating this model and providing the most reasonable solutions to validate a set's consistency. To check the complete implementations of these, you can find them in this repository.

Update the look-up table

A lookup table needs to be created with all existing email addresses. A row in this lookup table should be built on the *AccountCreatedEvent* and the *EmailAddressChangedEvent* with the accountId and the email address. You can do this by using an event handling component:

```
package io.axoniq.dev.samples.command.handler;

import java.util.Optional;

import org.axonframework.config.ProcessingGroup;
import org.axonframework.eventhandling.EventHandler;
import org.springframework.stereotype.Component;

import io.axoniq.dev.samples.api.AccountCreatedEvent;
import io.axoniq.dev.samples.api.EmailAddressChangedEvent;
import io.axoniq.dev.samples.command.persistence.EmailJpaEntity;
import io.axoniq.dev.samples.command.persistence.EmailRepository;


/**
 * Subscribing procesor that updates lookup table with email addresses
used in the Account.
 * Links to "Update the look-up table" in the blog
 *
 * @author Yvonne Ceelie
 */
@Component
@ProcessingGroup("emailEntity")
public class AccountEventHandler {

    @EventHandler
    public void on(AccountCreatedEvent event, EmailRepository
emailRepository) {
        emailRepository.save(new EmailJpaEntity(event.getEmailAddress(),
event.getAccountId()));
    }

    @EventHandler
    public void on(EmailAddressChangedEvent event, EmailRepository
emailRepository) {
        Optional<EmailJpaEntity> emailEntityOptional =
emailRepository.findEmailJpaEntityByAccountId(event.getAccountId());
    }
}
```

You'll need to make this processing group subscribe to update the lookup table in the same thread as the event has been applied by adding this property to your application.properties file:

```
axon.eventhandling.processors.emailEntity.mode=subscribing
```

Subscribing processors will create and maintain a projection immediately after an event has been applied and are directly consistent. Therefore, these lookup tables are always owned by the command side only and should not be exposed by using a Query API on top of it.

Now that you have a lookup table filled with all existing email addresses, you can actually *validate* if the email address exists before applying the event.

Validation through a Dispatch Interceptor

You can validate a command even before an Aggregate is loaded or created using a command dispatch interceptor. A dispatch interceptor intercepts the command message before handling it, allowing us to introduce the required validation. The following interceptor reacts on the *CreateAccountCommand* specifically, allowing it to validate the email set:

```
package io.axoniq.dev.samples.command.interceptor;
import io.axoniq.dev.samples.api.CreateAccountCommand;
import io.axoniq.dev.samples.command.persistence.EmailRepository;
import org.axonframework.commandhandling.CommandMessage;
import org.axonframework.messaging.MessageDispatchInterceptor;
import org.springframework.stereotype.Component;
import java.util.List;
import java.util.function.BiFunction;
/**
 * Intercepts the create account command message and throws
IllegalStateException when already account aggregate with
 * email address already exists. Links to "Validation through a Dispatch
Interceptor' section in in this [set based
 * validation blog](https://axoniq.io/blog-overview/set-based-validation)
 *
 * @author Yvonne Ceelie
 */
@Component
public class AccountCreationDispatchInterceptor implements
MessageDispatchInterceptor<CommandMessage<?>> {

    private final EmailRepository emailRepository;

    public AccountCreationDispatchInterceptor(EmailRepository
emailRepository) {
        this.emailRepository = emailRepository;
    }

    @Override
    public BiFunction<Integer, CommandMessage<?>, CommandMessage<?>>
handle(List<? extends CommandMessage<?>> list) {
        return (i, m) -> {
            if (CreateAccountCommand.class.equals(m.getPayloadType())) {
                final CreateAccountCommand createAccountCommand =
(CreateAccountCommand) m.getPayload();
                if
(emailRepository.existsById(createAccountCommand.getEmailAddress())) {
                    throw new
IllegalStateException(String.format("Account with email address %s
already exists",

createAccountCommand.getEmailAddress()));
                }
            }
            return m;
        };
    }
}
```

Validation in an External Command Handler

If an account already exists and the user wants to change the email address, a *RequestEmailChangeCommand* can be sent to a command handler outside the Aggregate. This command handler is typically called a "Command Handling Component" or an "External Command Handler". This command handler will validate the email set whenever it handles the RequestEmailChangeCommand:

```
package io.axoniq.dev.samples.command.handler;

import io.axoniq.dev.samples.api.ChangeEmailAddressCommand;
import io.axoniq.dev.samples.api.RequestEmailChangeCommand;
import io.axoniq.dev.samples.command.persistence.EmailRepository;
import org.axonframework.commandhandling.CommandHandler;
import org.axonframework.commandhandling.gateway.CommandGateway;
import org.springframework.stereotype.Component;

/**
 * Command handling component that validates if the email address already
exists. Links to the `Validation in an
 * External Command Handler` section in this [set based validation blog]
(https://axoniq.io/blog-overview/set-based-validation)
 *
 * @author Yvonne Ceelie
 */
@Component
public class AccountCommandHandler {

    @CommandHandler
    public void handle(RequestEmailChangeCommand command, CommandGateway
commandGateway,
                       EmailRepository emailRepository) {
        if (emailRepository.existsById(command.getUpdatedEmailAddress()))
{
            throw new IllegalStateException(String.format("Account with
email address %s already exists",

command.getUpdatedEmailAddress()));
        }
        commandGateway.send(new
ChangeEmailAddressCommand(command.getAccountId(),
command.getUpdatedEmailAddress()));
    }
}
```

Validation using a Parameter Resolver

The last note worth the solution is to introduce a “ParameterResolver”. The ParameterResolver can build such that it can resolve any parameter, including a Boolean value which is true only if the email already exists:

```
package io.axoniq.dev.samples.resolver;

import io.axoniq.dev.samples.api.ChangeEmailAddressCommand;
import io.axoniq.dev.samples.command.persistence.EmailRepository;
import org.axonframework.messaging.Message;
import org.axonframework.messaging.annotation.ParameterResolver;
import org.axonframework.messaging.annotation.ParameterResolverFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import java.lang.reflect.Executable;
import java.lang.reflect.Parameter;

/**
 * This parameter resolver resolves to true if an account aggregate with
email address already exists. Links to
 * "Validation using a Parameter Resolver" section in this [set based
validation blog](https://axoniq.io/blog-overview/set-based-validation)
 *
 * @author Yvonne Ceelie
 */
@Component
public class EmailAlreadyExistsResolverFactory implements
ParameterResolver<Boolean>, ParameterResolverFactory {

    private final EmailRepository emailRepository;

    @Autowired
    public EmailAlreadyExistsResolverFactory(EmailRepository
emailRepository) {
        this.emailRepository = emailRepository;
    }

    @Override
    public ParameterResolver createInstance(Executable executable,
Parameter[] parameters, int parameterIndex) {
        if (Boolean.class.equals(parameters[parameterIndex].getType())) {
            return this;
        }
        return null;
    }

    @Override
    public Boolean resolveParameterValue(Message<?> message) {
        if (matches(message)) {
            String emailAddress = ((ChangeEmailAddressCommand)
message.getPayload()).getUpdatedEmailAddress();
            return emailRepository.findById(emailAddress).isPresent();
        }
        throw new IllegalArgumentException("Message payload not of type
ChangeEmailAddressCommand");
    }

    @Override
    public boolean matches(Message<?> message) {
        return
ChangeEmailAddressCommand.class.isAssignableFrom(message.getPayloadType());
    }
    @Override
    public Class<?> supportedPayloadType() {
        return ChangeEmailAddressCommand.class;
    }
}
```

With this resolver in place, you can add the boolean email that already exists to the ChangeEmailAddressCommand command handler, merging the validation into your command model:

```
@CommandHandler
public void handle(ChangeEmailAddressCommand command, Boolean
emailAddressExists) {
    if (emailAddressExists) {
        throw new IllegalStateException(String.format("Account with email
address %s already exists",

command.getUpdatedEmailAddress()));
    }
    if (emailAddress.equals(command.getUpdatedEmailAddress())) {
        throw new IllegalStateException(String.format("Email address %s
is already used for account with id %s ",

command.getUpdatedEmailAddress(), accountId));
    }
    apply(new EmailAddressChangedEvent(command.getAccountId(),
command.getUpdatedEmailAddress()));
}
```

Conclusion

When separating an application in a command and query handling side, you must be aware the query handling side is eventually consistent. You cannot base decisions on the command handling side by using the query side because the query model may not be synchronized yet with the changes made on the command handling side. We can expand upon the command model by introducing an additional table next to the aggregate to resolve this. To update this extra table consistently in Axon, you can use a subscribing event processor to create and update it. This look-up table should only be used by the command handling side to base its decisions upon when validating commands. To use this table seamlessly within Axon, you can, for example, provide a Command Handler Interceptor, an External Command Handler, or a Parameter Resolver.

Many thanks to my colleague Steven van Beelen who helped me writing this blog.

Also, check out the podcast 'Exploring Axon', *where I talked about this subject (and more) with my colleague Sara Torrey.*

DDD, CQRS, Event Sourcing, Axon Framework