Table of Contents

Long-Running User Flows (10 pages)

> In the long run, the pessimist may be proven right, but the optimist has a better time on the trip.
>
> — Daniel Reardon

In the previous chapters, we have looked at handling commands and queries within the context of a single aggregate. All the scenarios we have looked at thus far, have been limited to a single interaction. However, not all capabilities can be implemented in the form of a simple request-response interaction, requiring coordination across multiple external systems or human-centric operations or both. In other cases, there may be a need to react to triggers that are nondeterministic (occur conditionally or not at all) and/or be time-bound (based on a deadline). This may require managing business transactions across multiple bounded contexts that may run over a long duration of time, while continuing to maintain consistency (**saga**).

There are at least two common patterns to implement the saga pattern:

- **Explicit orchestration**: A designated component acts as a centralized coordinator — where the system relies on the coordinator to react to domain events to manage the flow.
- **Implicit choreography**: No single component is required to act as a centralized coordinator — where the components simply react to domain events in other components to manage the flow.

By the end of this chapter, you will have learned how to implement sagas using both techniques. You will also have learnt how to handle exceptions using retries, compensating actions and deadlines. You will finally be able to appreciate when/whether to choose an explicit orchestrator or

simply stick to implicit choreography without resorting to the use of potentially expensive distributed transactions.

Technical requirements

- JDK 1.8+ (We have used Java 17 to compile sample sources)
- Spring Boot 2.4.x
- Axon framework 4.5.3
- JUnit 5.7.x (Included with spring boot)
- Project Lombok (To reduce verbosity)
- Maven 3.x

Continuing our design journey

In Chapter 4 - Domain analysis and modeling, we discussed eventstorming as a lightweight method to clarify business flows. As a reminder, this is the output produced from our eventstorming session:

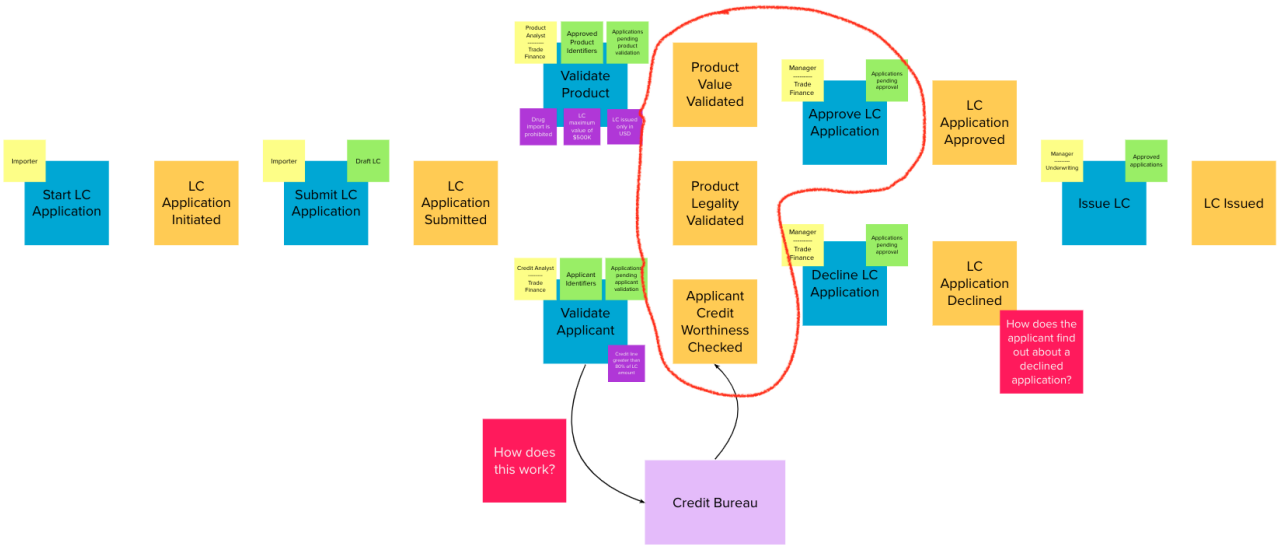

Figure 1. Recap of eventstorming session

As depicted in the visual above, some aspects of Letter of Credit (LC) application processing happens outside our current bounded context), before the trade finance manager makes a decision to either approve or decline the application as listed here:

1. Product value is validated for correctness
2. Product legality is validated
3. Applicant's credit worthiness is validated

Currently, the final approval is a manual process. It is pertinent to note that the product value and legality checks happen as part of the work done by the product analysis department, whereas applicant credit worthiness checks happens in the credit analysis department. Both departments make use of their own systems to perform these functions and notify us through the respective

events. An LC application is **not ready** to either be approved or declined until **each** of these checks are completed. Each of these processes happen mostly independently of the other and may take a nondeterministic amount of time (typically in the order of a few days). After these checks have happened, the trade finance manager manually reviews the application and makes the final decision.

Given the growing volumes of LC applications received, the bank is looking to introduce a process optimization to automatically approve applications with an amount below a certain threshold (**USD 10,000** at this time). The business has deemed that the three checks above are sufficient and that no further human intervention is required when approving such applications.

From an overall system perspective, it is pertinent to note that the product analyst system notifies us through the `ProductValueValidatedEvent` and `ProductLegalityValidatedEvent`, whereas the credit analyst system does the same through the `ApplicantCreditValidatedEvent` event. Each of these events can and indeed happen independently of the other. For us to be able to auto-approve applications our solution needs to wait for all of these events to occur. Once these events have occurred, we need to examine the outcome of each of these events to finally make a decision.

In this context, we are using the term ***long-running*** to denote a complex business process that takes several steps to complete. As these steps occur, the process transitions from one state to another. In other words, we are referring to a **state machine**[1]. This is not to be confused with a long-running software process (for example, a complex SQL query or an image processing routine) that is computationally intensive.

As is evident, the LC auto-approval functionality is an example of a long-running business process where *some thing* in our system needs to keep track of the fact that these independent events have occurred before proceeding further. Such functionality can be implemented using the saga pattern. Let's look at how we can do this.

Implementing sagas

Before we delve into how we can implement this auto-approval functionality, let's take a look at how this works from a logical perspective as shown here:

[auto approval saga logical] | *sagas/auto-approval-saga-logical.png*

Figure 2. Auto-approval process — logical view

As is depicted in the visual above, there are three bounded contexts in play:

1. LC Application (the bounded context we have been implementing thus far)
2. The Applicant bounded context
3. The Product bounded context

The flow gets triggered when the LC application is submitted. This in turn sets in motion three independent functions that establish the:

1. Value of the product being transacted

2. Legality of the product being transacted
3. Credit worthiness of the applicant

LC approval can proceed only after **all** of these functions have completed. Furthermore, to **auto-approve**, all of these checks have to complete **favorably** and as mentioned earlier, the LC amount has to be lesser than the **USD 10000** threshold.

As shown in the event storming artifact, the `LC Application` aggregate is able to handle an `ApproveLCApplicationCommand`, which results in a LCApplicationApprovedEvent`. To auto-approve, this command needs to be invoked automatically when all the conditions mentioned earlier are satisfied. We are building an event-driven system, and we can see that each of these validations produce events when their respective actions complete. There are at least two ways to implement this functionality:

1. Orchestration: where a single component in the system coordinates the state of the flow and triggers subsequent actions as necessary.
2. Choreography: where actions in the flow are triggered without requiring an explicit coordinating component.

Let's examine these methods in more detail:

Orchestration

When implementing sagas using an orchestrating component, the system looks similar to the one depicted here:

[auto approval saga orchestrator] | *sagas/auto-approval-saga-orchestrator.png*

Figure 3. Saga implementation using an orchestrator

The orchestrator starts tracking the flow when the LC application is submitted. It will then need to wait for each of the `ProductValueValidatedEvent`, `ProductLegalityValidatedEvent` and `ApplicantCreditValidatedEvent` events to occur and decide if it is appropriate to trigger the `ApproveLCApplicationCommand`. Finally, the saga lifecycle ends unconditionally when the LC application is approved. There are other conditions that may cause the saga to end abruptly. We will examine those scenarios in detail later. It is pertinent to note that there will be a **distinct** auto-approval saga instance for each LC application that gets submitted. Let's look at how to implement this functionality using the Axon framework:

```
import org.axonframework.modelling.saga.EndSaga;
import org.axonframework.modelling.saga.SagaEventHandler;
import org.axonframework.modelling.saga.StartSaga;
import org.axonframework.spring.stereotype.Saga;

@Saga                                                                    ①
public class AutoApprovalSaga {

    @SagaEventHandler(associationProperty = "lcApplicationId") ②
    public void on(ApplicantCreditValidatedEvent event) {
        //
    }

    @SagaEventHandler(associationProperty = "lcApplicationId") ②
    public void on(ProductValueValidatedEvent event) {
        //
    }

    @SagaEventHandler(associationProperty = "lcApplicationId") ②
    public void on(ProductLegalityValidatedEvent event) {
        //
    }

    @SagaEventHandler(associationProperty = "lcApplicationId") ②
    @StartSaga                                                           ③
    public void on(LCApplicationSubmittedEvent event) {
        //
    }

    @SagaEventHandler(associationProperty = "lcApplicationId") ③
    @EndSaga                                                             ④
    public void on(LCApplicationApprovedEvent event) {
        //
    }
}
```

① When working with Axon and Spring, the orchestrator is annotated with the `@Saga` annotation to mark it as a spring bean. In order to track each submitted LC application, the `@Saga` annotation is prototype-scoped (as opposed to singleton-scoped), to allow creation of multiple saga instances. Please refer to the Axon and Spring documentation for more information.

② The saga listens to relevant events happening in the system to keep track of the flow (as denoted by the `@SagaEventHandler` annotation). Conceptually, the `@SagaEventHandler` annotation is very similar to the `@EventHandler` annotation that we discussed previously. However, in contrast to the `@EventHandler` which are singletons, a distinct instance of the saga is created for each LC application. This distinction is established by the value of the `associationProperty` attribute. In this example, the `associationProperty` looks for the `lcApplicationId` attribute in the incoming event's payload.

③ In the case of the `AutoApprovalSaga`, the `LCApplicationSubmittedEvent` marks the beginning of the

saga lifecycle (as denoted by the `@StartSaga` annotation).

④ The saga ends unconditionally when the `LCApplicationApprovedEvent` occurs (as denoted by the `@EndSaga` annotation). As mentioned previously, it is possible to end sagas conditionally as well. We will look at how we can do this later in this section.

The Axon framework uses the term **saga** for its orchestrator-based implementation, whereas we have used the term saga to denote the pattern of tracking long-running transactions.

Choreography

Orchestration versus choreography

Handling distributed exceptions

Types of exceptions

Business exceptions

System exceptions

Recovery

Automated recovery

Manual recovery

Compensating actions

Retries

Handling deadlines

Summary

Questions

Further reading

Title	Author	Location
Example	Author	https://www.example.com
Example	Author	https://www.example.com

[1] https://en.wikipedia.org/wiki/state_machine