Table of Contents

Long-Running User Flows (10 pages) . . . 1
- Technical requirements . . . 2
- Continuing our design journey . . . 2
- Implementing sagas . . . 3
  - Orchestration . . . 3
  - Choreography . . . 3
  - Orchestration versus choreography . . . 3
- Handling distributed exceptions . . . 3
  - Types of exceptions . . . 3
  - Recovery . . . 4
- Handling deadlines . . . 4
- Summary . . . 4
- Questions . . . 4
- Further reading . . . 4

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:

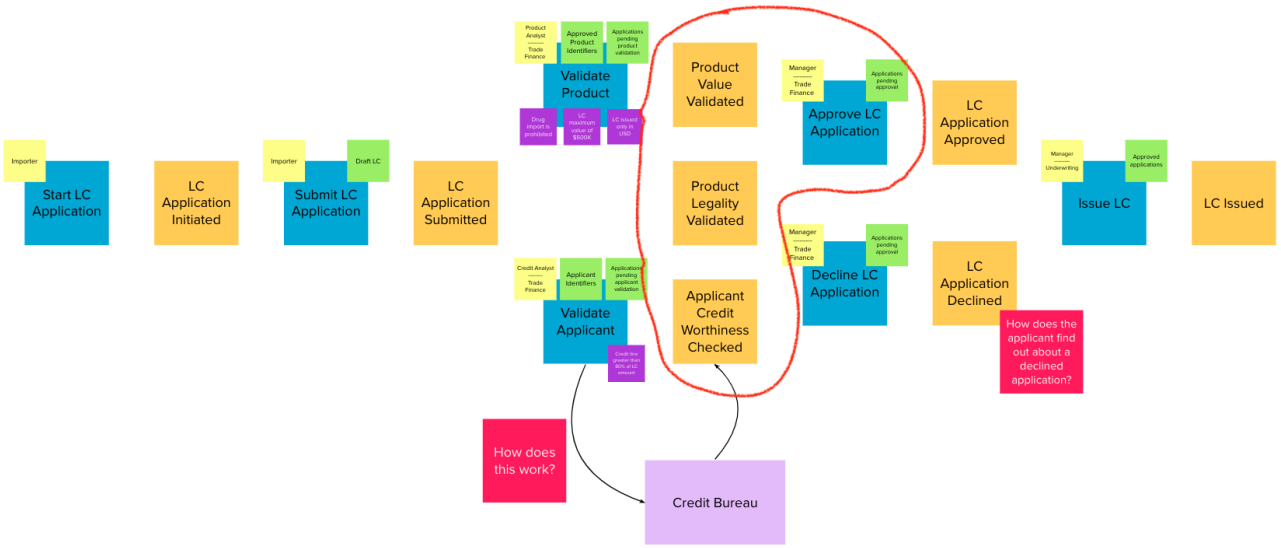

Figure 1. Recap of eventstorming session

As depicted in the visual above, Letter of Credit (LC) applications go through the following validations before the trade finance manager makes a decision to either approve or decline the application:

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

Currently, this 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.

Implementing sagas

Orchestration

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