Table of Contents

Long-Running Workflows

> 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 work with deadlines when no explicit events occur within the system. 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

To follow the examples in this chapter, you will need access to:

- 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

Please refer to the ch08 directory of the book's accompanying source code repository for complete working examples.

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:

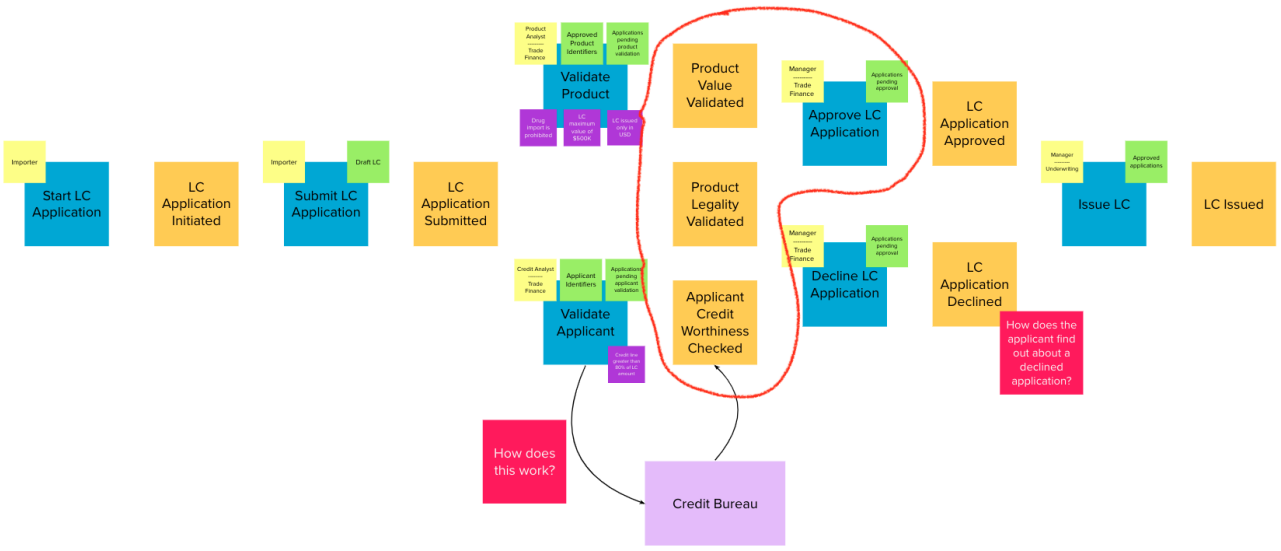

Figure 1. Recap of eventstorming session

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
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 from the diagram above, 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:

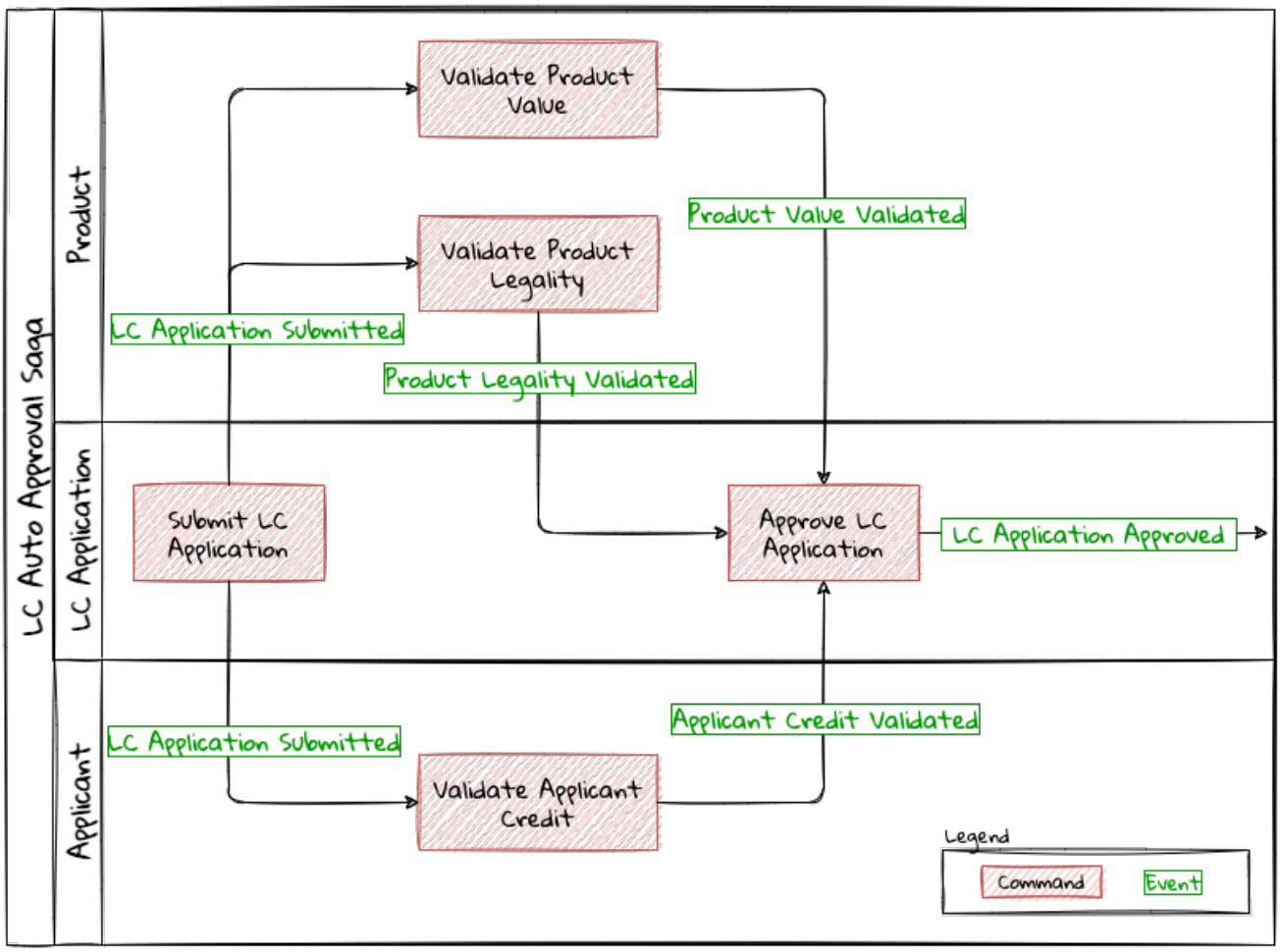

Figure 2. Auto-approval process — logical view

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:

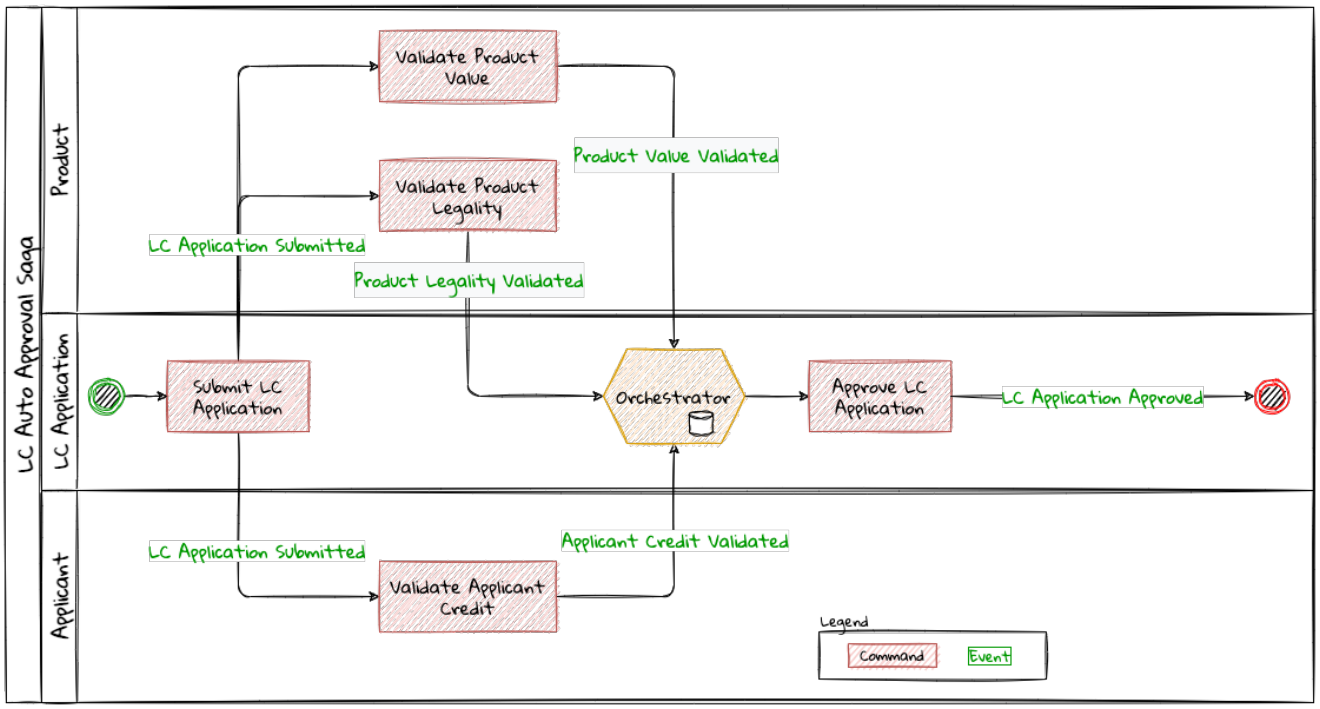

Figure 3. Saga implementation using an orchestrator

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. As usual, let's test drive this functionality that a new auto approval saga instance is created when an LC application is submitted:

```
import org.axonframework.test.saga.FixtureConfiguration;
import org.axonframework.test.saga.SagaTestFixture;

class AutoApprovalSagaTests {

    private FixtureConfiguration fixture;                                    ①

    @BeforeEach
    void setUp() {
        fixture = new SagaTestFixture<>(AutoApprovalSaga.class);             ①
    }

    @Test
    void shouldStartSagaOnSubmit() {
        final LCApplicationId lcApplicationId = LCApplicationId.randomId();
        fixture.givenNoPriorActivity()                                       ②
                .whenPublishingA(                                            ③
                        new LCApplicationSubmittedEvent(lcApplicationId,
                            AUTO_APPROVAL_THRESHOLD_AMOUNT
                               .subtract(ONE_DOLLAR)))
                .expectActiveSagas(1);                                       ④
    }

}
```

① We make use of the Axon provided `FixtureConfiguration` and `SagaTestFixture` that allow us to test saga functionality.

② Given no prior activity has occurred (from the perspective of the saga)

③ When a `LCApplicationSubmittedEvent` is published

④ We expect one active saga to exist

The implementation to make this test pass looks like:

```
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
    @StartSaga                                                       ③
    public void on(LCApplicationSubmittedEvent 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 the `LCApplicationSubmittedEvent` 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 in Chapter 7. However, the `@SagaEventHandler` annotation is used specifically for event listeners within a saga. The `associationProperty` attribute on the `@SagaEventHandler` annotation causes this event handler method to get invoked only for the saga with matching value of the `lcApplicationId` attribute in the event payload. Also, the `@SagaEventHandler` is a transaction boundary. Every time such a method completes successfully, the Axon framework commits a transaction, thereby allowing it to keep track of state stored in the saga. We will look at this in more detail shortly.

③ Every saga needs to have at least one `@SagaEventHandler` method that is also annotated with the `@StartSaga` annotation to denote the beginning of the saga.

We have a requirement that an LC cannot be auto-approved if its amount exceeds the threshold (USD 10000 in our case). The test for this scenario looks like:

```
class AutoApprovalSagaTests {
    //...

    @Test
    void shouldEndSagaImmediatelyIfAmountGreaterThanAutoApprovalThreshold() {
        final LCApplicationId lcApplicationId = LCApplicationId.randomId();
        fixture.givenAggregate(lcApplicationId.toString()).published()
                .whenPublishingA(
                        new LCApplicationSubmittedEvent(lcApplicationId,
                            AUTO_APPROVAL_THRESHOLD_AMOUNT.add(ONE_DOLLAR))) ①
                .expectActiveSagas(0);                                       ②
    }
}
```

① When the LC amount exceeds the auto approval threshold amount

② We expect no active sagas to exist for that LC

The implementation to satisfy this condition looks like this:

```
import org.axonframework.modelling.saga.SagaLifecycle;

@Saga
public class AutoApprovalSaga {

    @SagaEventHandler(associationProperty = "lcApplicationId")
    @StartSaga
    public void on(LCApplicationSubmittedEvent event) {
        if (AUTO_APPROVAL_THRESHOLD_AMOUNT.isLessThan(event.getAmount())) { ①
            SagaLifecycle.end();                                            ②
        }
    }
}
```

① We check for the condition of the LC amount being greater than the threshold amount

② If so, we end the saga using the framework provided `SagaLifecycle.end()` method. Here we end the saga programmatically. It is also possible to declaratively end the saga as well using the `@EndSaga` annotation when the `LCApplicationApprovedEvent` occurs. Please refer to the full code examples included with this chapter for more information.

We need to auto-approve the saga if all of `ApplicantCreditValidatedEvent`, `ProductLegalityValidatedEvent` and `ProductValueValidatedEvent` have occurred successfully. The test to verify this functionality is shown here:

```
class AutoApprovalSagaTests {

    @Test
    void shouldAutoApprove() {
        // Initialization code removed for brevity

        fixture.givenAggregate(lcApplicationId.toString())
            .published(submitted, legalityValidated, valueValidated)        ①
                .whenPublishingA(applicantValidated)                        ②
                .expectActiveSagas(1)                                       ③
                .expectDispatchedCommands(
                        new ApproveLCApplicationCommand(lcApplicationId));  ④
    }
}
```

① Given that the LC application has been submitted and the `ProductValueValidatedEvent` and the `ProductLegalityValidatedEvent` have occurred successfully.

② When the `ApplicantCreditValidatedEvent` is published

③ We expect one active saga instance AND

④ We expect the `ApproveLCApplicationCommand` to be dispatched for that LC

The implementation for this looks like:

```
class AutoApprovalSaga {

    private boolean productValueValidated;                                  ①
    private boolean productLegalityValidated;                               ①
    private boolean applicantValidated;                                     ①

    @Autowired
    private transient CommandGateway gateway;                               ②

    // Other event handlers omitted for brevity

    @SagaEventHandler(associationProperty = "lcApplicationId")
    public void on(ApplicantCreditValidatedEvent event) {                   ③
        if (event.getDecision().isRejected()) {                             ④
            SagaLifecycle.end();
        } else {
            this.applicantValidated = true;                                 ⑤
            if (productValueValidated && productLegalityValidated) {        ⑥
                LCApplicationId id = event.getLcApplicationId();
                gateway.send(ApproveLCApplicationCommand.with(id));         ⑦
            }
        }
    }

    // Other event handlers omitted for brevity
}
```

① As mentioned previously, sagas can maintain state. In this case, we are maintaining three boolean variables, each to denote the occurrence of the respective event.

② We have declared the Axon `CommandGateway` as a `transient` member because we need it to dispatch commands, but not be persisted along with other saga state.

③ This event handler intercepts the `ApplicantCreditValidatedEvent` for the specific LC application (as denoted by the `associationProperty` in the @SagaEventHandler annotation).

④ If the decision from the `ApplicantCreditValidatedEvent` is rejected, we end the saga immediately.

⑤ Otherwise, we *remember* the fact that the applicant's credit has been validated.

⑥ We then check to see if the product's value and legality have already been validated.

⑦ If so, we issue the command to auto-approve the LC.

The logic in the `ProductValueValidatedEvent` and `ProductLegalityValidatedEvent` is very similar to that in the saga event handler for the `ApplicantCreditValidatedEvent`. We have omitted it here for brevity. Please refer to the source code for this chapter for the full example along with the tests.

Finally, we can end the saga when we receive the LCApplicationApprovedEvent for this application.

```
class AutoApprovalSagaTests {
    @Test
    @DisplayName("should end saga after auto approval")
    void shouldEndSagaAfterAutoApproval() {
        // Initialization code omitted for brevity

        fixture.givenAggregate(lcApplicationId.toString())
                .published(
                    submitted, applicantValidated,
                    legalityValidated, valueValidated)                          ①
                .whenPublishingA(new LCApplicationApprovedEvent(lcApplicationId)) ②
                .expectActiveSagas(0)                                           ③
                .expectNoDispatchedCommands();                                  ④
    }
}
```

① Given that the LC has been submitted and all the validations have been completed successfully.

② When a `LCApplicationApprovedEvent` is published.

③ We expect zero active sagas to be running.

④ And we also expect to not dispatch any commands.

Now that we have looked at how to implement sagas using an orchestrator, let's examine some design decisions that we may need to consider when working with them.

Pros

- **Complex workflows**: Having an explicit orchestrator can be very helpful when dealing with flows that involve multiple participants and have a lot of conditionals because the orchestrator can keep track of the overall progress in a fine-grained manner.
- **Testing**: As we have seen in the implementation above, testing flow logic in isolation is relatively straightforward.
- **Debugging**: Given that we have a single coordinator, debugging the current state of the flow can be relatively easier.
- **Handling exceptions**: Given that the orchestrator has fine-grained control of the flow, recovering gracefully from exceptions can be easier.
- **System knowledge**: Components in different bounded contexts do not need to have knowledge of each other's internals (e.g. commands and events) to progress the flow.
- **Cyclic dependencies**: Having a central coordinator allows avoiding accidental cyclic dependencies between components.

Cons

- **Single point of failure**: From an operational perspective, orchestrators can become single points of failure because they are the only ones that have knowledge of the flow. This means that these components need to exhibit higher resilience characteristics as compared to other components.

- **Leaking of domain logic**: In an ideal world, the aggregate will remain the custodian of all domain logic. Given that the orchestrator is also stateful, business logic may inadvertently shift to the orchestrator. Care should be taken to ensure that the orchestrator only has flow-control logic while business invariants remains within the confines of the aggregate.

The above implementation should give you a good idea of how to implement a saga orchestrator. Now let's look at how we can do this without the use of an explicit orchestrator.

Choreography

Saga orchestrators keep track of the current state of the flow, usually making use of some kind of data store. Another way to implement this functionality is without using any stateful component. Logically, this looks like the setup shown in the diagram here:

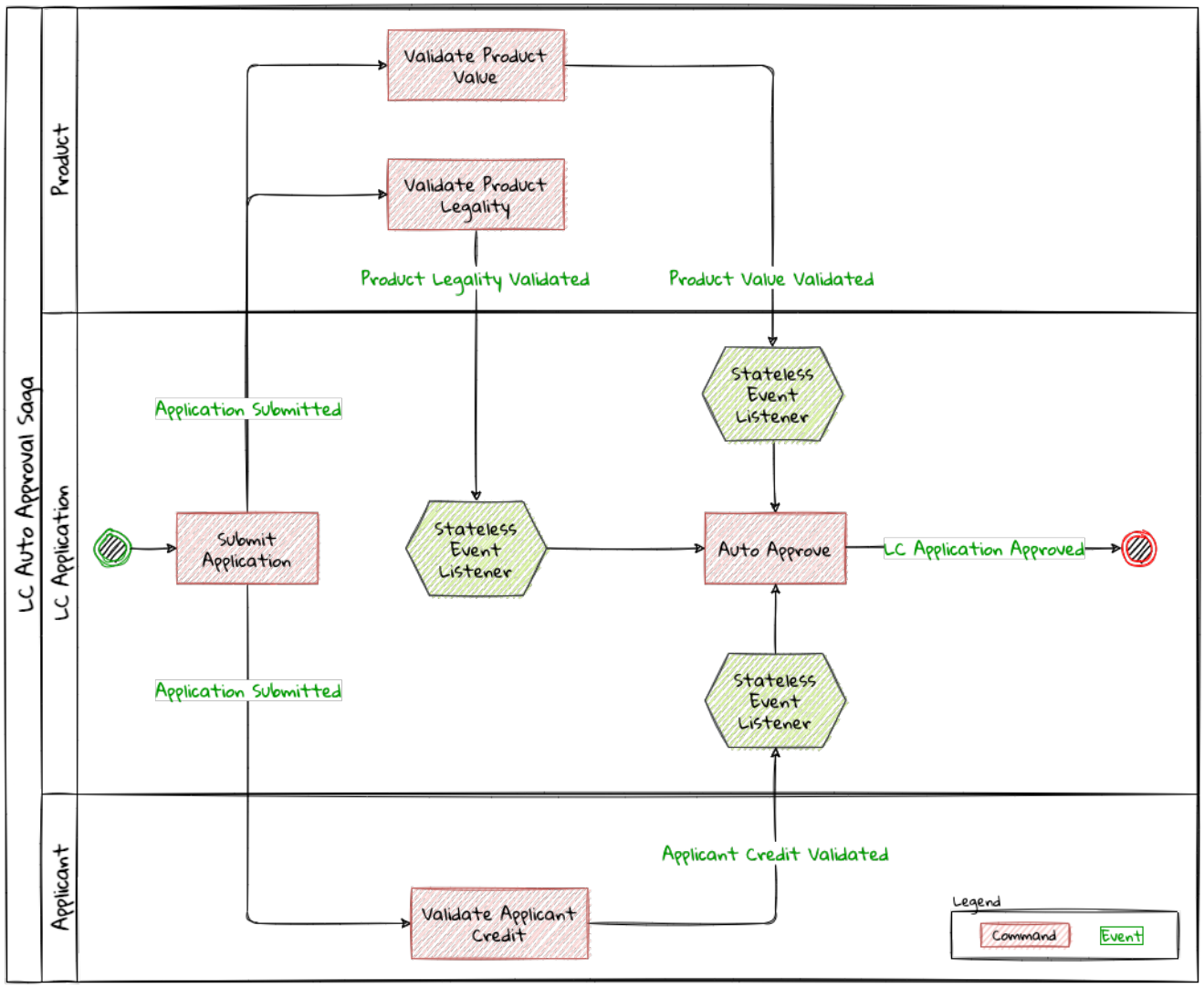

Figure 4. Saga implementation using choreography

As you can see, there is no single component that tracks the saga lifecycle. However, to make the auto-approval decision, each of these stateless event handlers need to have knowledge of the same three events having occurred:

1. Product value is validated
2. Product legality is validated

3. Applicant's credit worthiness is validated

Given that the event listeners themselves are stateless, there are at least two ways to provide this information to them:

1. Each of the events carry this information in their respective payloads.
2. The event listeners query the source systems (in this case, the product and applicant bounded contexts respectively).
3. The LC application bounded context maintains a query model to keep track of these events having occurred.

Just like in the orchestrator example, when all events have occurred and the LC amount is below the specified threshold, these event listeners can issue the `ApproveLCApplicationCommand`.

We will skip covering code examples for the choreography implementation because this is no different from the material we have covered previously in this and prior chapters.

Now that we have looked at how to implement the choreography style of sagas, let's examine some design decisions that we may need to consider when working with them.

Pros

- **Simple workflows**: For simple flows, the choreography approach can be relatively straightforward because it does not require the overhead of an additional coordinating component.
- **No single points of failure**: From an operational perspective, there is one less high resilience component to worry about.

Cons

- **Workflow tracking**: Especially with complex workflows that involve numerous steps and conditionals, tracking and debugging the current state of the flow may become challenging.
- **Cyclic dependencies**: It is possible to inadvertently introduce cyclic dependencies among components when workflows become gnarly.

Sagas enable applications to maintain data and transactional consistency when more than one bounded context is required to complete the business functionality without having to resort to using *distributed transactions*[2]. However, it does introduce a level of complexity to the programming model, especially when it comes to handling failures. We will look at exception handling in a lot more detail when we discuss working with distributed systems in upcoming chapters. Let's look at how to progress flows when there are no explicit stimuli by looking at how deadlines work.

Handling deadlines

Thus far, we have looked at events that are caused by human (for example, the applicant

submitting an LC application) or system (for example, the auto-approval of an LC application) action. However, in an event-driven system, not all events occur due to an explicit human or system stimulus. Events may need to be emitted either due to inactivity over a period of time, or on a recurring schedule based on prevailing conditions.

For example, let's examine the case where the bank needs *submitted LC applications* to be decisioned as quickly as possible. When applications are not acted upon by the trade finance managers within ten calendar days, the system should send them reminders.

To deal with such inactivity, we need a means to trigger system action (read — emit events) based on the passage of time — in other words, perform actions when a *deadline* expires. In a happy path scenario, we expect either the user or the system to take certain action. In such cases, we will also need to account for cases we will need to cancel the trigger scheduled to occur on deadline expiry. Let's look at how to test-drive this functionality.

```
class LCApplicationAggregateTests {
    //...
    @Test
    void shouldCreateSubmissionReminderDeadlineWhenApplicationIsSubmitted() {
        final LCApplicationId id = LCApplicationId.randomId();
        fixture.given(new LCApplicationStartedEvent(id, ApplicantId.randomId(),
                                "My LC", LCState.DRAFT),
                        new LCAmountChangedEvent(id, THOUSAND_DOLLARS),
                        new MerchandiseChangedEvent(id, merchandise()))

                .when(new SubmitLCApplicationCommand(id)) ①
                .expectEvents(new LCApplicationSubmittedEvent(id,
                                THOUSAND_DOLLARS))

                .expectScheduledDeadlineWithName(
                        Duration.ofDays(10),
                        LC_APPROVAL_PENDING_REMINDER);    ②
    }
}
```

① When the LC application is submitted

② We expect a deadline for the reminder to be scheduled

The implementation for this is fairly straightforward:

```
import org.axonframework.deadline.DeadlineManager;

class LCApplication {
    //...
    @CommandHandler
    public void on(SubmitLCApplicationCommand command,
                   DeadlineManager deadlineManager) { ①
        assertPositive(amount);
        assertMerchandise(merchandise);
        assertInDraft(state);
        apply(new LCApplicationSubmittedEvent(id, amount));

        deadlineManager.schedule(Duration.ofDays(10),  ②
            "LC_APPROVAL_REMINDER",
            LCApprovalPendingNotification.first(id));  ③
    }
    //...
}
```

① To allow working with deadlines, the Axon framework provides a `DeadlineManager` that allows working with deadlines. This is injected into the command handler method.

② We use the `deadlineManager` to schedule a named deadline (`"LC_APPROVAL_REMINDER"` in this case) that will expire in 10 days.

③ When the deadline is met, it will result in a `LCApprovalPendingNotification` which can be handled just like a command. Except in this case, the behavior is triggered by the passage of time.

If no action is taken for ten days, this is what we expect:

```
class LCApplication {

    @Test
    void shouldTriggerApprovalPendingEventTenDaysAfterSubmission() {
        final LCApplicationId id = LCApplicationId.randomId();
        fixture.given(new LCApplicationStartedEvent(id, ApplicantId.randomId(),
                                "My LC", LCState.DRAFT),
                        new LCAmountChangedEvent(id, THOUSAND_DOLLARS),
                        new MerchandiseChangedEvent(id, merchandise()))
                .andGivenCommands(new SubmitLCApplicationCommand(id)) ①
                .whenThenTimeElapses(Duration.ofDays(10))             ②
                .expectDeadlinesMet(
                        LCApprovalPendingNotification.first(id))      ③
                .expectEvents(new LCApprovalPendingEvent(id));        ④
    }
}
```

① Given that the LC application is submitted.

② When the period of ten days elapses.

③ The deadline should be met.

④ And the `LCApprovalPendingEvent` should be emitted.

Let's look at how to implement this:

```
import org.axonframework.deadline.annotation.DeadlineHandler;

class LCApplication {

    @DeadlineHandler(deadlineName = "LC_APPROVAL_REMINDER")       ①
    public void on(LCApprovalPendingNotification notification) {  ②

        AggregateLifecycle.apply(new LCApprovalPendingEvent(id)); ③

    }
}
```

① Deadlines are handled by annotating handler methods with the `@DeadlineHandler` annotation. Note that the same deadline name used previously is being referenced here.

② This is the deadline handler method and uses the same payload that was passed along when it was scheduled.

③ We emit the `LCApprovalPendingEvent` when the deadline expires.

The deadline handling logic should only be triggered if no action is taken. However, if the LC is either approved or rejected within a duration of ten days, none of this behavior should be triggered:

```
class LCApplicationAggregateTests {
    //...
    @Test
    void shouldNotTriggerPendingReminderIfApplicationIsApprovedWithinTenDays() {
        final LCApplicationId id = LCApplicationId.randomId();
        fixture.given(new LCApplicationStartedEvent(id, ApplicantId.randomId(),
                                    "My LC", LCState.DRAFT),
                            new LCAmountChangedEvent(id, THOUSAND_DOLLARS),
                            new MerchandiseChangedEvent(id, merchandise()))
                .andGivenCommands(new SubmitLCApplicationCommand(id)) ①

                .when(new ApproveLCApplicationCommand(id))            ②
                .expectEvents(new LCApplicationApprovedEvent(id))
                .expectNoScheduledDeadlines();                        ③
    }

    @Test
    void shouldNotTriggerPendingReminderIfApplicationIsDeclinedWithinTenDays() {
        // Test code is very similar. Excluded for brevity
    }

}
```

① Given that the LC application is submitted

② When it is approved within a duration of ten days (in this case, almost immediately)

③ We expect no scheduled deadlines

And the implementation for this looks like:

```
class LCApplication {
    //...
    @CommandHandler
    public void on(ApproveLCApplicationCommand command,
                   DeadlineManager deadlineManager) {
        assertInSubmitted(state);
        AggregateLifecycle.apply(new LCApplicationApprovedEvent(id));
        deadlineManager.cancelAllWithinScope("LC_APPROVAL_REMINDER"); ①
    }

    @CommandHandler
    public void on(DeclineLCApplicationCommand command,
                   DeadlineManager deadlineManager) {
        assertInSubmitted(state);
        AggregateLifecycle.apply(new LCApplicationDeclinedEvent(id));
        deadlineManager.cancelAllWithinScope("LC_APPROVAL_REMINDER"); ①
    }

    //...
}
```

① We cancel all the deadlines with the name `LC_APPROVAL_REMINDER` (in this case, we only have one deadline with that name) within the scope of this aggregate.

Summary

In this chapter, we examined how to work with long-running workflows using sagas and the different styles we can use to implement them. We also looked at the implications of using explicit orchestration versus implicit choreography. We finally looked at how we can handle deadlines when there are no user-initiated actions.

You should have learnt how sagas can act as a first-class citizen in addition to aggregates when designing a system that makes use of domain-driven design principles.

In the next chapter, we will look at how we can interact with external systems while respecting bounded context boundaries between core and peripheral systems.

Questions

- Are you seeing the need to implement long-running workflows in your current ecosystem?
- Do you see yourself picking one style over the other most of the time?
- Are you using external deadline handlers (for instance, batch jobs) in your existing systems as opposed to embedding time-based logic in the core?

Further reading

Title	Author	Location
Saga persistence and event-driven architectures	Udi Dahan	https://udidahan.com/2009/04/20/saga-persistence-and-event-driven-architectures/
Sagas solve stupid transaction timeouts	Udi Dahan	https://udidahan.com/2008/06/23/sagas-solve-stupid-transaction-timeouts/
Microservices — when to react vs. orchestrate	Andrew Bonham	https://medium.com/capital-one-tech/microservices-when-to-react-vs-orchestrate-c6b18308a14c
Saga orchestration for microservices using the outbox pattern	Gunnar Morling	https://www.infoq.com/articles/saga-orchestration-outbox/
Patterns for distributed transactions within a microservices architecture	Keyang Xiang	https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture

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

[2] https://en.wikipedia.org/wiki/Distributed_transaction