Saga Orchestration for Microservices Using the Outbox Pattern

This item in japanese

Lire ce contenu en franÃ§ais

Key Takeaways

- Sagas allow for the implementation of long-running, distributed business transactions, executing a set of operations across multiple microservices, applying consistent all-or-nothing semantics.
- For the sake of decoupling, communication between microservices should preferably happen asynchronously, for instance using distributed commit logs like Apache Kafka.
- The outbox pattern provides a solution for service authors to perform writes to their local database and send messages via Apache Kafka, without relying on unsafe “dual writes.”
- Debezium, a distributed open-source change data capture platform, provides a robust and flexible foundation for orchestrating Saga flows using the outbox pattern.

When moving to microservices, one of the first things to realize is that individual services don’t exist in isolation. While the goal is to create loosely coupled, independent services with as little interaction as possible, chances are high that one service needs a particular data set owned by another service, or that multiple services need to act in concert to achieve a consistent outcome of an operation in the domain of our business.

The outbox pattern, implemented via change data capture, is a proven approach for addressing the concern of data exchange between microservices; avoiding any unsafe "dual writes" to multiple resources, e.g., a database and a message broker, the outbox pattern achieves eventually consistent data exchange, without depending on the synchronous availability of all participants, and not requiring complex protocols such as XA (a widely used standard for distributed transaction processing defined by The Open Group) either.

In this article, I'd like to explore how to take the outbox pattern to the next level and use it for implementing Sagas—potentially long-running business transactions that span across multiple microservices. One common example is that of booking a trip comprising multiple parts: either all flight legs and accommodation should be booked together or none of them. Sagas split up one such overarching business transaction into a series of multiple local database transactions, that are executed by the participating services.

A Primer on Sagas

For "rolling back" the overarching business transaction in case of a failure, Sagas rely on the notion of compensating transactions: each previously applied local transaction must be able to be "undone" by running another transaction which applies the inversion of the formerly done changes.

Sagas are by no means a new concept—they have first been discussed by Hector Garcia-Molina and Kenneth Salem in their SIGMOD '87 Sagas paper. But in the context of the continuing move towards microservices architectures, Sagas backed by local transactions within the participating services are seeing growing popularity these days, as indicated by the MicroProfile specification for Long Running Actions, which is currently under active development. Sagas lend themselves in particular to the implementation of transactional flows taking a longer period of time, which typically can't be addressed with ACID semantics.

To make things tangible, let's consider the example of an e-commerce business with three services: order, customer, and payment. When a new purchase order is submitted to the order service, the following flow should be executed—including the other two services:

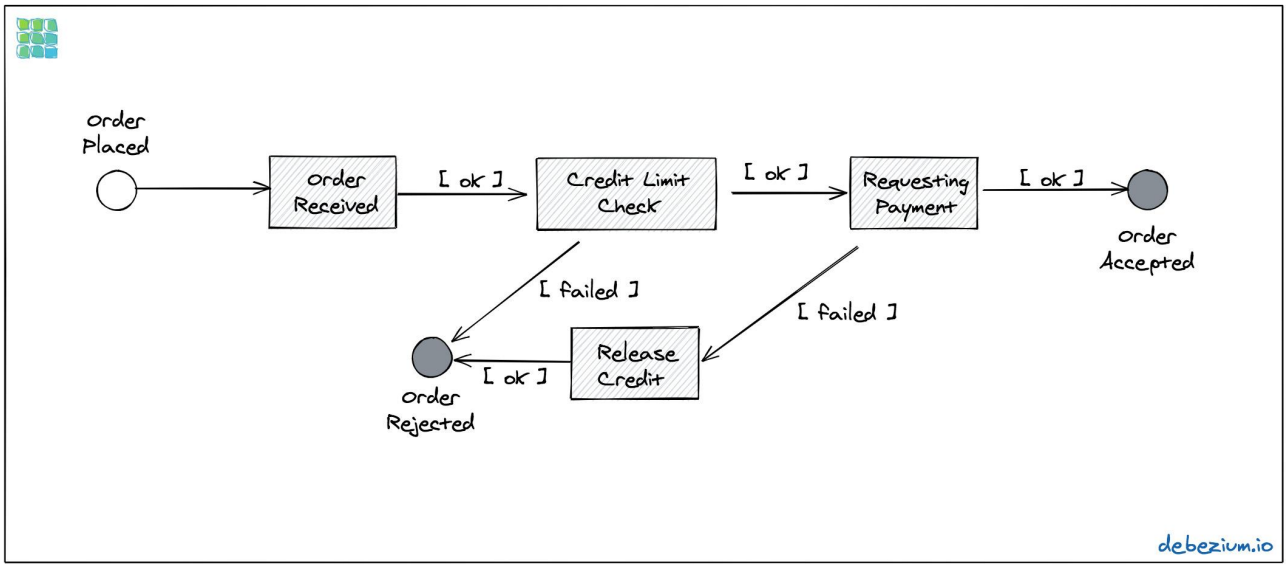


Figure 1. Order state transitions

First, we need to check with the customer service whether the incoming order fits into the customer's credit limit (since we don't want to have pending orders of one customer to pass a certain threshold). If the customer has a credit limit of $500, and a new order with a value of $300 comes in, then this order would fit into the current limit with a remaining limit of $200. A subsequent order with a value of $250 would be rejected accordingly, as it would exceed the customer's then-current open credit limit.

If the credit limit check succeeds, the payment for the order needs to be requested via the payment service. If both the credit limit check and the payment request succeed, the order transitions into the `Accepted` state, so fulfillment for it could begin (which isn't part of the process discussed here).

If the credit limit check fails, however, the order will go to the `Rejected` state right away. If that step succeeds but the subsequent payment request fails, the previously allocated credit limit needs to be released again, before transitioning to the `Rejected` state.

Implementation Choices

There are two general ways for implementing distributed Sagas—choreography and orchestration. In the choreography approach, one participating service sends a message to the next after it has executed its local transaction. With orchestration, on the other hand, there's one coordinating service that invokes one participant after the other.

Both approaches have their pros and cons, (e.g., see this post by Chris Richardson and this one by Yves do Régo for a more detailed discussion). Personally, I prefer the orchestration approach, as it defines one central place that can be queried to obtain the current status of a particular Saga (the orchestrator, or "Saga execution coordinator," SEC for short). Since it avoids point-to-point communication between participants, (other than the orchestrator), it also allows for the addition of further intermediary steps within the flow, without the need to adjust each participant.

Before diving into the implementation of such Saga flow, it's worth spending some time to think about the transactional semantics that Sagas provide. Let's examine how Sagas satisfy the four classic ACID properties of transactions, as defined by Theo Härder and Andreas Reuter (based on earlier work by Jim Gray) in their fundamental paper Principles of Transaction-Oriented Database Recovery:

- ***A**tomicity*: ✅ —The pattern ensures that either all services apply the local transactions or, in case of a failure, all already executed local transactions are compensated so that no data change is applied effectively.
- ***C**onsistency*: ✅ —All local constraints are guaranteed to be satisfied after successful execution of all the transactions making up the Saga, transitioning the overall system from one consistent state to another.
- ***I**solation*: ❌ —As local transactions are committed while the Saga is running, their changes are already visible to other concurrent transactions, despite the possibility that the Saga will fail eventually, causing all previously applied transactions to be compensated. I.e., from the perspective of the overall Saga, the isolation level is comparable to "read uncommitted."
- ***D**urability*: ✅ —Once the local transactions of the Saga have been committed, their changes are persisted and durable, e.g., after a service failure and restart.

From a service consumer point of view—e.g., a user placing a purchase order with the order service—the system is eventually consistent; i.e., it will take some time until the purchase order is in its correct state, as per the logic of the different participating services.

As far as the communication between the participating services is concerned, this may happen either synchronously, e.g., via HTTP or gRPC, or asynchronously, e.g., via message brokers or distributed logs such as Apache Kafka. Whenever possible, asynchronous communication between services should be preferred, as it unties the sending service from the availability of consuming ones. And as we'll see in the next section, not even the availability of Kafka itself is a concern, thanks to change data capture.

Recap: The Outbox Pattern

Now, how do the outbox pattern and change data capture (as provided by Debezium) fit into all this? As said above, a Saga coordinator should preferably communicate asynchronously with participating services, via request and reply message channels. Apache Kafka is a super-popular choice for implementing these channels. But the orchestrator (and each participating service) also need to apply transactions to their specific databases to execute their parts of the overall Saga flow.

While it might be tempting to simply execute a database transaction and send a corresponding message to Kafka shortly thereafter, this isn't a good idea. These two actions wouldn't happen within a single transaction spanning the database and Kafka. It'll be only a matter of time until we end up with an inconsistent state when, e.g., the database transaction commits but the write to Kafka fails. But friends don't let friends do dual writes, and the outbox pattern provides a very elegant way for addressing this issue:

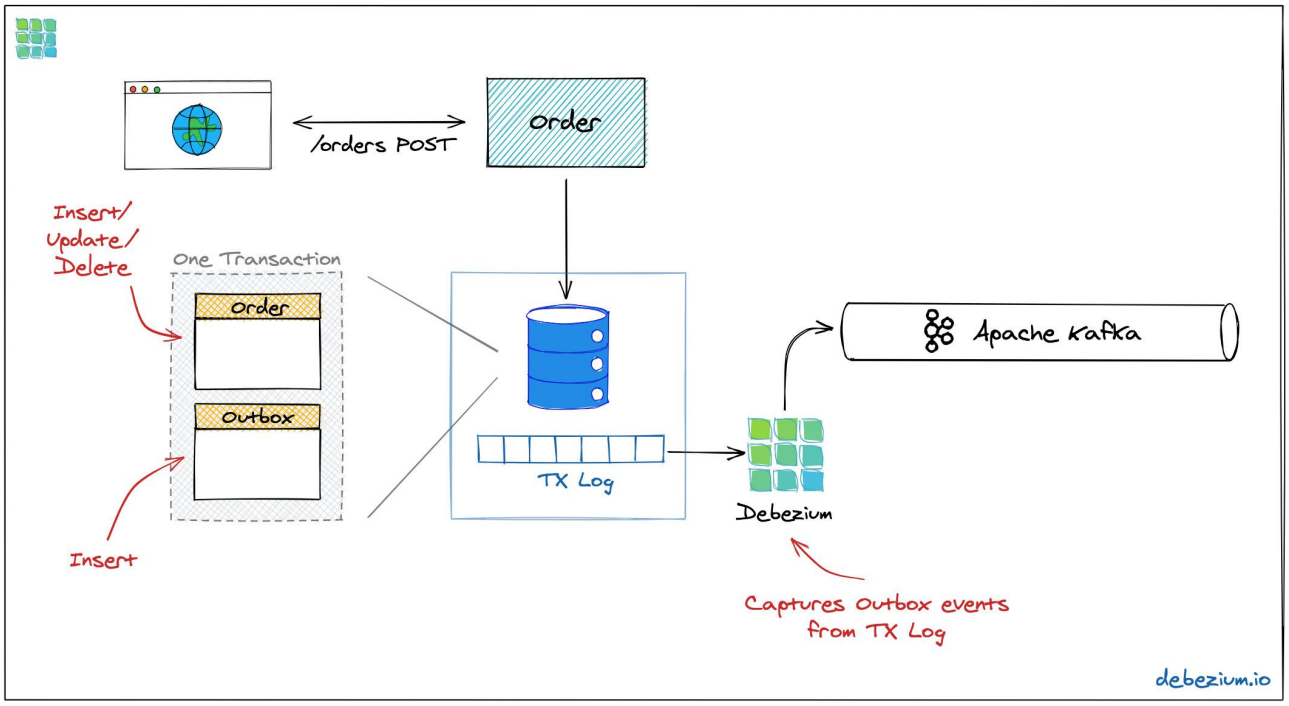


Figure 2. Safely updating the database and sending a message to Kafka via the outbox pattern

Instead of directly sending a message to Kafka when updating the database, the service uses a single transaction to both perform the normal update and insert the message into a specific outbox table within its database. Because this is done within a single database transaction, either the changes to the service's model are persisted and the message gets safely stored in the outbox table, or none of these changes gets applied. Once the transaction has been written to the database's transaction log, the Debezium change data capture process can pick up the outbox message from there and send it to Apache Kafka.

This is done using at-least-once semantics: under specific circumstances, the same outbox message could be sent to Kafka multiple times. To allow consumers to detect and ignore duplicate messages, each message should have a unique id. This could for instance be a UUID or a monotonically increasing sequence specific to each message producer, propagated as a Kafka message header.

Implementing Sagas Using the Outbox Pattern

With the outbox pattern in our toolbox, things become a bit clearer. The order service, acting as the Saga coordinator, triggers the entire flow after an incoming order placement call (typically via a REST API), by updating its local state—comprising the persisted order model and the Saga execution log—and emits messages to the other two participating services, one after another.

These two services react to the messages which they receive via Kafka, perform a local transaction that updates their data state and emit a reply message for the coordinator via their outbox table. The overall solution design looks like this:

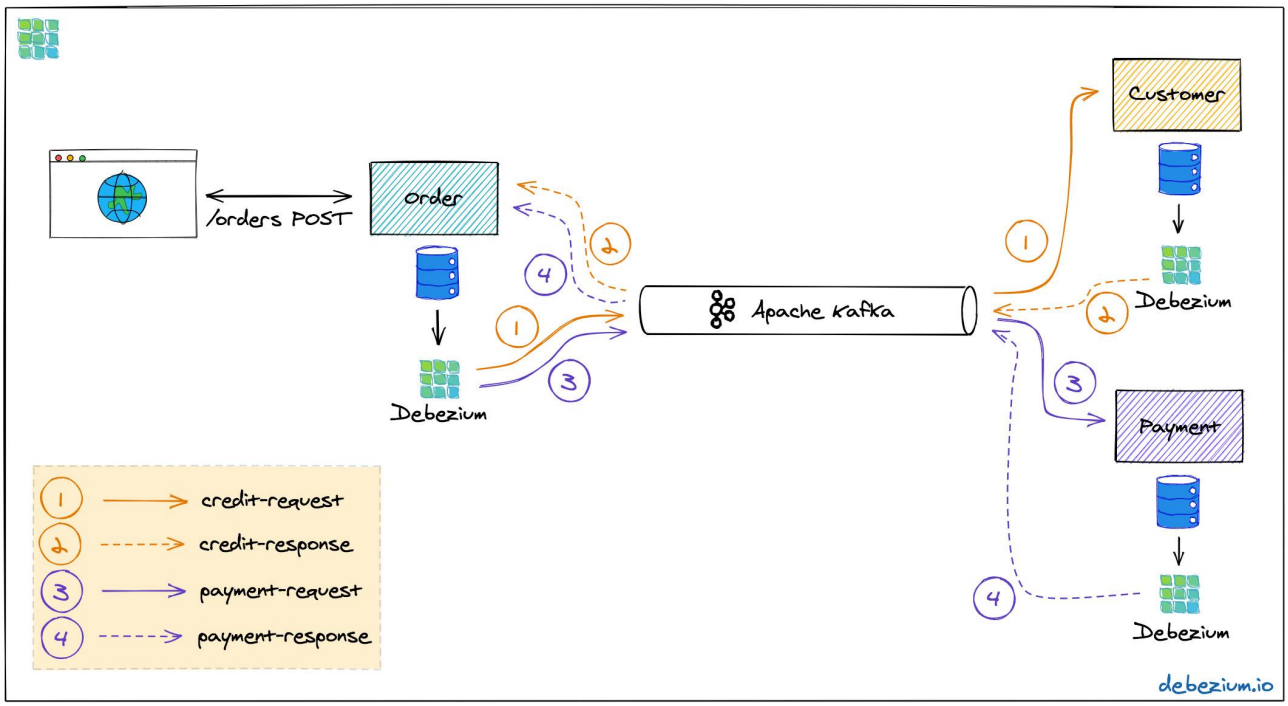


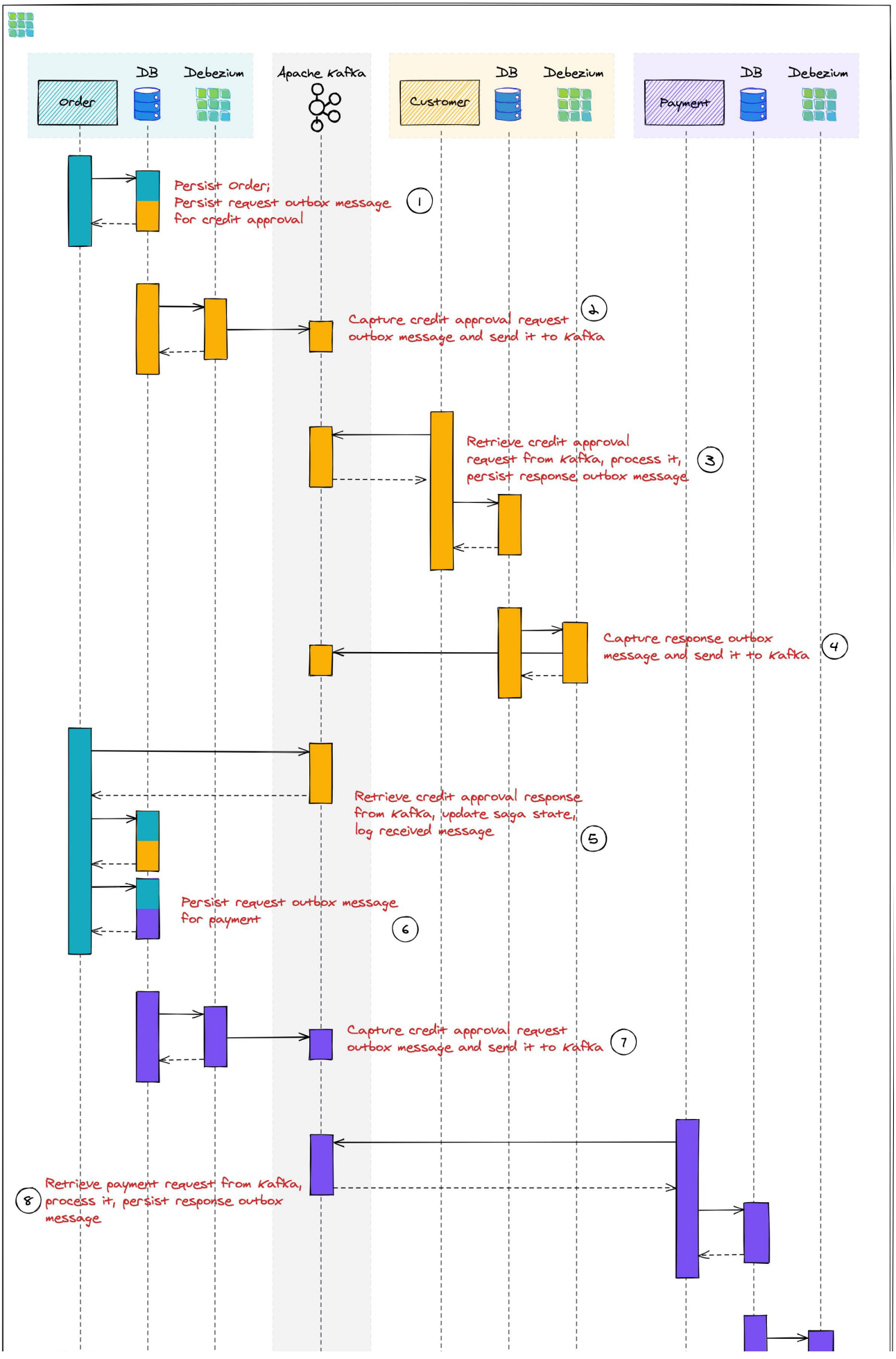
Figure 3. Saga orchestration using the outbox pattern

You can find a complete proof-of-concept implementation of this architecture in the Debezium examples repository on GitHub. The key parts of the architecture are these:

- The three services, order (for managing purchase orders and acting as the Saga orchestrator), customer (for managing the customer's credit limit), and payment (for handling credit card payments), each with their own local database (Postgres)
- Apache Kafka as the messaging backbone
- Debezium, running on top of Kafka Connect, subscribing to changes in the three different databases, and sending them to corresponding Kafka topics, using Debezium's outbox event routing component

The three services are implemented using Quarkus, a stack for building cloud-native microservices either running on the JVM or compiled down to native binaries (via GraalVM). Of course, the pattern could also be implemented using other stacks or languages, as long as they provide a means of consuming messages from Kafka and writing to a database. Also, a combination of different implementation technologies is possible.

There are four Kafka topics involved: a request and a response topic for the credit approval messages, and a request and a response topic for the payment messages. In case of a successful Saga execution, exactly four messages would be exchanged. If one of the steps fails and a compensating transaction is necessary, there'd be additional pairs of request and response messages for each step to be compensated.


Order
DB
Debezium
Apache Kafka
Customer
DB
Debezium
Payment
DB
Debezium
Persist order;
Persist request outbox message
for credit approval
1
Capture credit approval request
outbox message and send it to kafka
2
Retrieve credit approval
request from kafka, process it,
persist response outbox message
3
Capture response outbox
message and send it to kafka
4
Retrieve credit approval response
from kafka, update saga state,
log received message
5
Persist request outbox message
for payment
6
Capture credit approval request
outbox message and send it to kafka
7
Retrieve payment request from kafka,
process it, persist response outbox
message
8

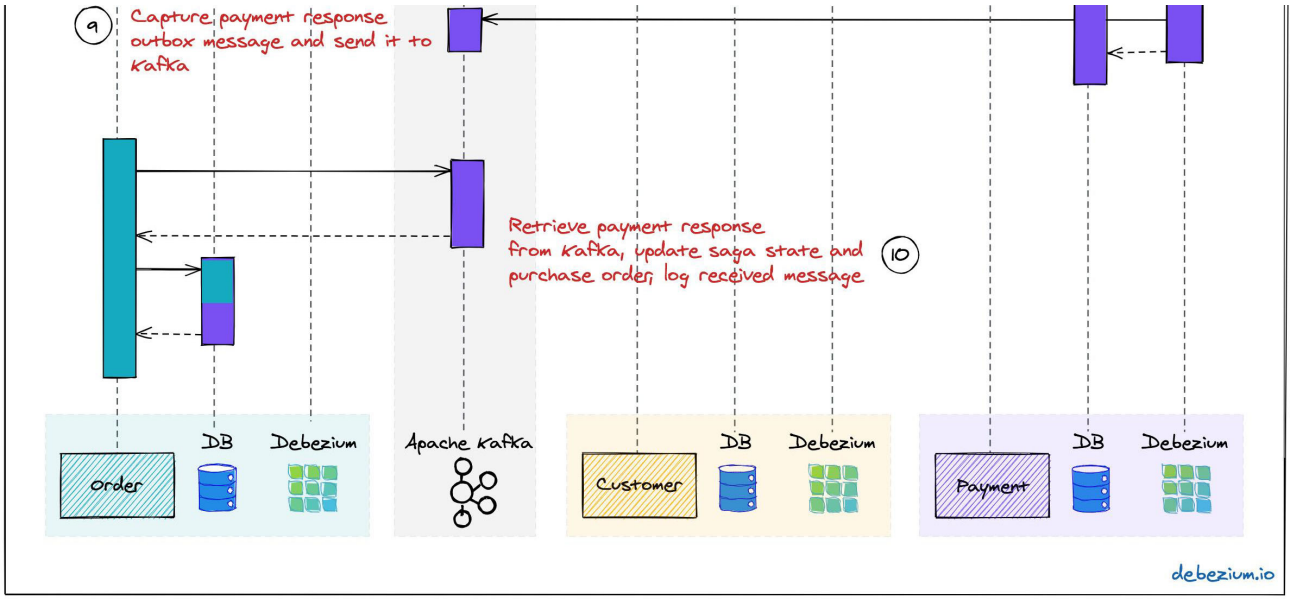


Figure 4. The execution sequence of a successful Saga flow

Each service emits outgoing messages via the outbox table in its own database. From there, the messages are captured via Debezium and sent to Kafka, and finally consumed by the receiving service. Upon sending and receiving messages, the order service, acting as the orchestrator, also persists the Saga progress in a local state table. (More on that below.) Furthermore, all participants log the ids of the messages they've consumed in a journal table, to identify potential duplicates later on.

Now, what happens if one step of the flow is failing? Let's assume the payment step fails because the customer's credit card has expired. In that case, the previously reserved credit amount in the customer service needs to be released again. To do so, the order service sends a compensation request to the customer service. Zooming out a bit (as the details around Debezium and Kafka are the same as before), this is what the message exchange would look like in this case:

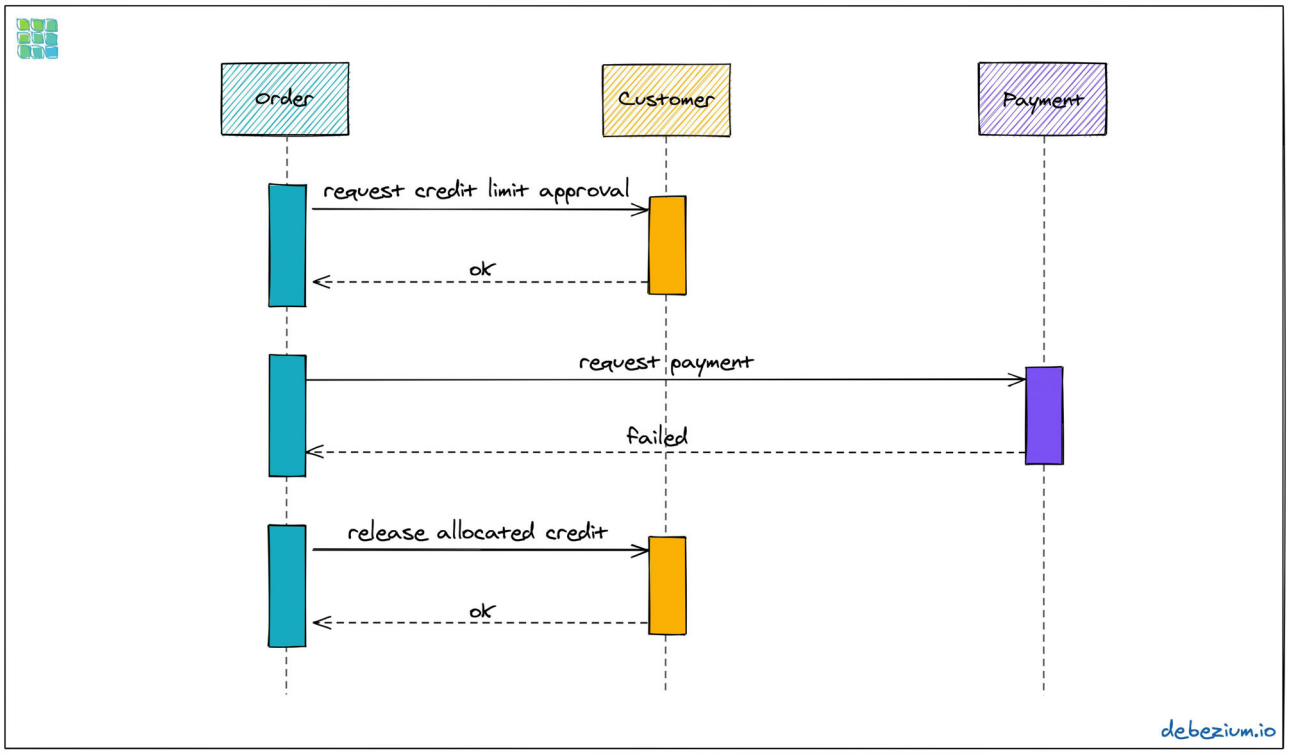


Figure 5. The execution sequence of a Saga flow with compensation

Having discussed the message flow between services, let's now dive into some implementation details of the order service. The proof-of-concept implementation provides a generic Saga orchestrator in the form of a simple state machine and the order-specific Saga implementation, which will be discussed in more depth further below. The "framework" part of the order service's implementation keeps track of the current state of the Saga execution within the `sagastate` table, whose schema looks like this:

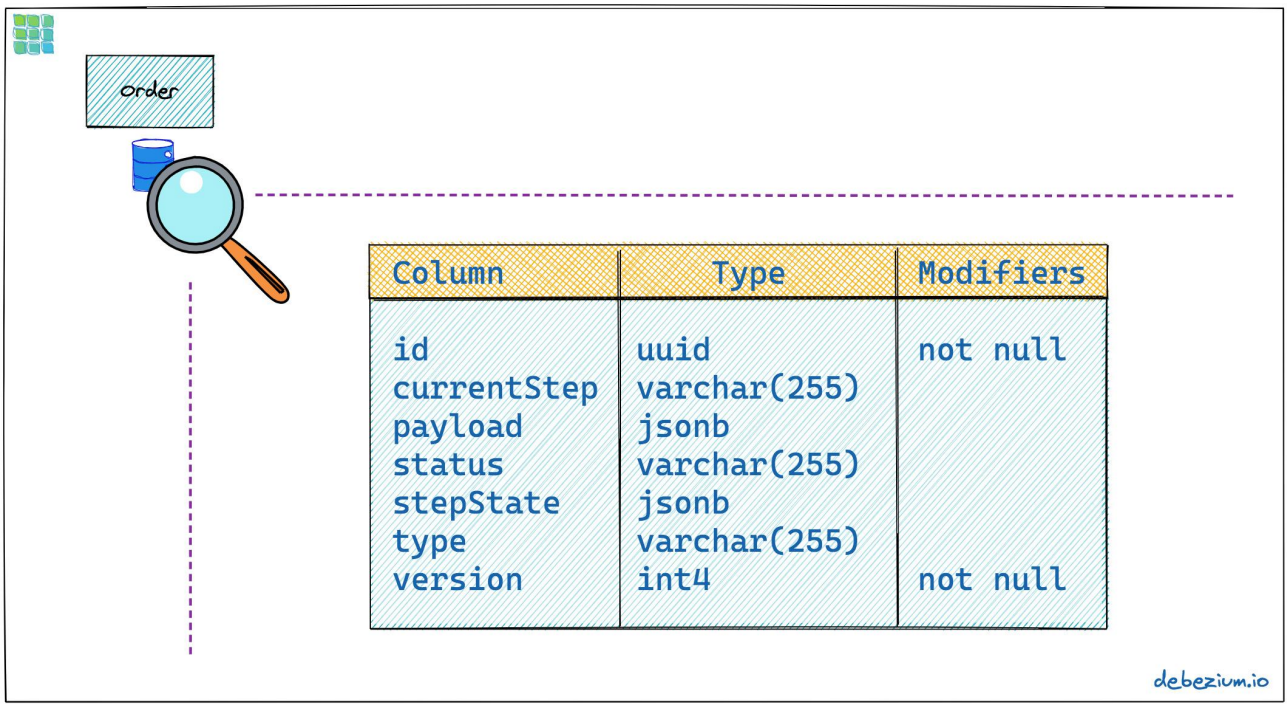


Figure 6. Schema of the Saga state table

This table fulfills the role of the Saga Log. Its columns are these:

- `id`
  : Unique identifier of a given Saga instance, representing the creation of one particular purchase order
- `currentStep`
  : The step at which the Saga currently is, e.g., “credit-approval” or “payment”
- `payload`
  : An arbitrary data structure associated with a particular Saga instance, e.g., containing the id of the corresponding purchase order and other information useful during the Saga lifecycle; while the example implementation uses JSON as the payload format, one could also think of using other formats, for instance, Apache Avro, with payload schemas stored in a schema registry
- `status`
  : The current status of the Saga; one of
  `STARTED, SUCCEEDED, ABORTING`
  , or
  `ABORTED`
- `stepState`
  : A stringified JSON structure describing the status of the individual steps, e.g.,
  `"{\"credit-approval\":\"SUCCEEDED\",\"payment\":\"STARTED\"}"`

- `type`
  : A nominal type of a Saga, e.g., “order-placement”; useful to tell apart different kinds of Sagas supported by one system
- `version`
  : An optimistic locking version, used to detect and reject concurrent updates to one Saga instance (in which case the message triggering the failing update needs to be retried, reloading the current state from the Saga log)

As the order service sends requests to the customer and payment services and receives their replies from Kafka, the Saga state gets updated within this table. By setting up a Debezium connector for tracking the `sagastate` table, we can nicely examine the progress of a Saga’s execution in Kafka.

Here are the state transitions for a purchase order whose payment fails; first, the order comes in and the “credit-approval” step gets started:


```
{
  "id": "73707ad2-0732-4592-b7e2-79b07c745e45",
  "currentstep": null,
  "payload": "\"order-id\": 2, \"customer-id\": 456, \"payment-due\"
  "sagastatus": "STARTED",
  "stepstatus": "{}",
  "type": "order-placement",
  "version": 0
}
{
  "id": "73707ad2-0732-4592-b7e2-79b07c745e45",
  "currentstep": "credit-approval",
  "payload": "{ \"order-id\": 2, \"customer-id\": 456, ... }",
  "sagastatus": "STARTED",
  "stepstatus": "{\"credit-approval\": \"STARTED\"}",
  "type": "order-placement",
  "version": 1
}
```

At this point, a “credit-approval” request message has been persisted in the outbox table, too. Once this has been sent to Kafka, the customer service will process it and send a reply message. The order service processes this by updating the Saga state and starting the payment step:


```
{
  "id": "73707ad2-0732-4592-b7e2-79b07c745e45",
  "currentstep": "payment",

  "payload": "{ \"order-id\": 2, \"customer-id\": 456, ... }",
  "sagastatus": "STARTED",
  "stepstatus": "{\"payment\": \"STARTED\", \"credit-approval\": \"S
  "type": "order-placement",
  "version": 2
}
```


Again a message is sent via the outbox table, now the “payment” request. This fails, and the payment system responds with a reply message indicating this fact. This means that the “credit-approval” step needs to be compensated via the customer system:


```
{
  "id": "73707ad2-0732-4592-b7e2-79b07c745e45",
  "currentstep": "credit-approval",
  "payload": "{ \"order-id\": 2, \"customer-id\": 456, ... }",
  "sagastatus": "ABORTING",
  "stepstatus": "{\"payment\": \"FAILED\", \"credit-approval\": \"CO
  "type": "order-placement",
  "version": 3
}
```


Once that has succeeded, the Saga is in its final state, `ABORTED`:


```
{
  "id": "73707ad2-0732-4592-b7e2-79b07c745e45",
  "currentstep": null,
  "payload": "{ \"order-id\": 2, \"customer-id\": 456, ... }",
  "sagastatus": "ABORTED",
```

```
  "sagastatus": "ABORTED",
  "stepstatus": "{\"payment\": \"FAILED\", \"credit-approval\": \"CO
  "type": "order-placement",
  "version": 4
}
```

You can try this out yourself by following the instructions in the example's README file, where you'll find requests for placing successful as well as failing order creations. It also has instructions for examining the exchanged messages in the Kafka topics sourced from the outbox tables of the different services.

Now let's look into some parts of the use case specific implementation. The Saga flow gets started within the order service's REST endpoint implementation like so:

```
@POST
@Transactional
public PlaceOrderResponse placeOrder(PlaceOrderRequest req) {
    PurchaseOrder order = req.toPurchaseOrder();
    order.persist();

    sagaManager.begin(OrderPlacementSaga.class, OrderPlacementSaga.p

    return PlaceOrderResponse.fromPurchaseOrder(order);
}
```

Persist the incoming purchase order

Begin the order placement Saga flow for the incoming order

`SagaMananger.begin()` will create a new record in the `sagastate` table, obtain the first outbox event from the `OrderPlacementSaga` implementation, and persist it in the outbox table. The `OrderPlacementSaga` class implements all the use case specific parts of the Saga flow:

- outbox events to be sent for executing one part of the Saga flow
- outbox events for compensating one part of the Saga flow
- event handlers for processing reply messages from the other Saga participants

The `OrderPlacementSaga` implementation is a tad too long for showing it here in its entirety, (you can find its complete source on GitHub), but here are some key parts:

```
@Saga(type="order-placement", stepIds = {CREDIT_APPROVAL, PAYMENT})
public class OrderPlacementSaga extends SagaBase {

  private static final String REQUEST = "REQUEST";
  private static final String CANCEL = "CANCEL";
  protected static final String PAYMENT = "payment";
  protected static final String CREDIT_APPROVAL = "credit-approval";


  // ...
  @Override
  public SagaStepMessage getStepMessage(String id) { 2
    if (id.equals(PAYMENT)) {
      return new SagaStepMessage(PAYMENT, REQUEST, getPayload());
    }
    else {
      return new SagaStepMessage(CREDIT_APPROVAL, REQUEST, getPayloa
    }
  }

  @Override
  public SagaStepMessage getCompensatingStepMessage(String id) { 3
    // ...
  }

  public void onPaymentEvent(PaymentEvent event) { 4
    if (alreadyProcessed(event.messageId)) {
      return;
    }

    onStepEvent(PAYMENT, event.status.toStepStatus());
    updateOrderStatus();

    processed(event.messageId);
  }
```

```
  public void onCreditApprovalEvent(CreditApprovalEvent event) { 5
     // ...
  }

  private void updateOrderStatus() { 6
    if (getStatus() == SagaStatus.COMPLETED) {
      PurchaseOrder order = PurchaseOrder.findById(getOrderId());
      order.status = PurchaseOrderStatus.ACCEPTED;
    }
    else if (getStatus() == SagaStatus.ABORTED) {

      PurchaseOrder order = PurchaseOrder.findById(getOrderId());
      order.status = PurchaseOrderStatus.CANCELLED;
    }
  }

  // ...
}
```

1 The ids of the Saga steps in order of execution

2 Returns the outbox message to be emitted for the given step

3 Returns the outbox message to be emitted for compensating the given step

4 Event handler for "payment" reply messages; it will update the purchase order status as well as the Saga status (via the onStepEvent() callback), which depending on the status may either complete the Saga or initiate its rollback by applying all the compensating messages

5 Event handler for "credit approval" reply messages

6 Updates the purchase order status, based on the current Saga states

...

```
this.outboxEvent.fire(CreditEvent.of(sagaId, CreditStatus.CANCELLED)
...
```

The implementation of customer and payment services isn't anything fundamentally new, so they are omitted here for the sake of brevity. You can find their complete source code here and here.

When Things Go Wrong

A key part of implementing distributed interaction patterns like Sagas is understanding how they behave in failure scenarios and making sure that (eventual) consistency is also achieved under such unforeseen circumstances.

Note that a negative outcome of any of the Saga steps (e.g., if the payment service rejects the payment due to an invalid credit card) isn't a failure scenario here; it is explicitly expected that participants can't successfully execute their part of the overall flow, resulting in the execution of appropriate compensating local transactions. This also means that such generally anticipated failure of execution must not result in a rollback of the local database transaction, as otherwise no reply message would be sent back to the orchestrator via the outbox.

With that in mind, let's discuss some possible failure scenarios:

The event handler of a Kafka message raises an exception

The local database transaction is rolled back and the message consumer does not acknowledge to the Kafka broker that it was able to process the message. Because the broker receives no confirmation that the message was processed, after some time it will resend the message repeatedly until it gets acknowledged. You should have monitoring in place to detect such a situation because the Saga flow won't be able to continue until the message has been processed.

The Debezium connector crashes after sending an outbox message to Kafka, but before committing the offset in the source database's transaction log

After restarting the connector, it will continue to read the messages from the outbox table beginning at the log offset that was committed last, potentially resulting in some outbox events sent a second time; that's why all the participants need to be idempotent, as implemented in the example using unique message ids and consumers tracking successfully processed messages via the journal tables.

The Kafka broker isn't running or can't be reached, for example, due to a network split

The Debezium connectors can resume their work after Kafka is available and accessible again; until then, Saga flows naturally can't proceed.

A message gets processed, but acknowledging it with Kafka fails

The message will be passed to the consuming service again, which would find the message's id in its journal table and thus ignore the duplicated message.

Concurrent updates to the Saga state table when processing multiple Saga steps in parallel

While we've discussed a sequential flow with the orchestrator triggering participating services one after another, you might also envision a Saga implementation that processes multiple steps in parallel. In this case, concurrently arriving reply messages may compete to update the Saga state table. This situation would be detected via the optimistic locking implemented on that table, causing an event handler trying to commit an update based on a superseded version of the Saga state to fail, rollback, and retry.

We could discuss some more cases, but the general semantics of the overall design are those of an eventually consistent system with at-least-once guarantees.

Bonus: Distributed Tracing

When designing an event flow between distributed systems, operational insight is vital for making sure everything runs correctly and efficiently. Distributed tracing provides such insights: it collects trace information from the individual systems that contribute to such interaction and allows examining the call flows, e.g., in a web UI, making it an invaluable tool for failure analysis and debugging

Debezium's outbox support addresses this concern through tight integration with the OpenTracing spec (support for OpenTelemetry is on the roadmap). By putting a tool such as Jaeger into place, it's just a matter of configuration to collect trace information from the order, customer, and payment services and display the end-to-end traces.

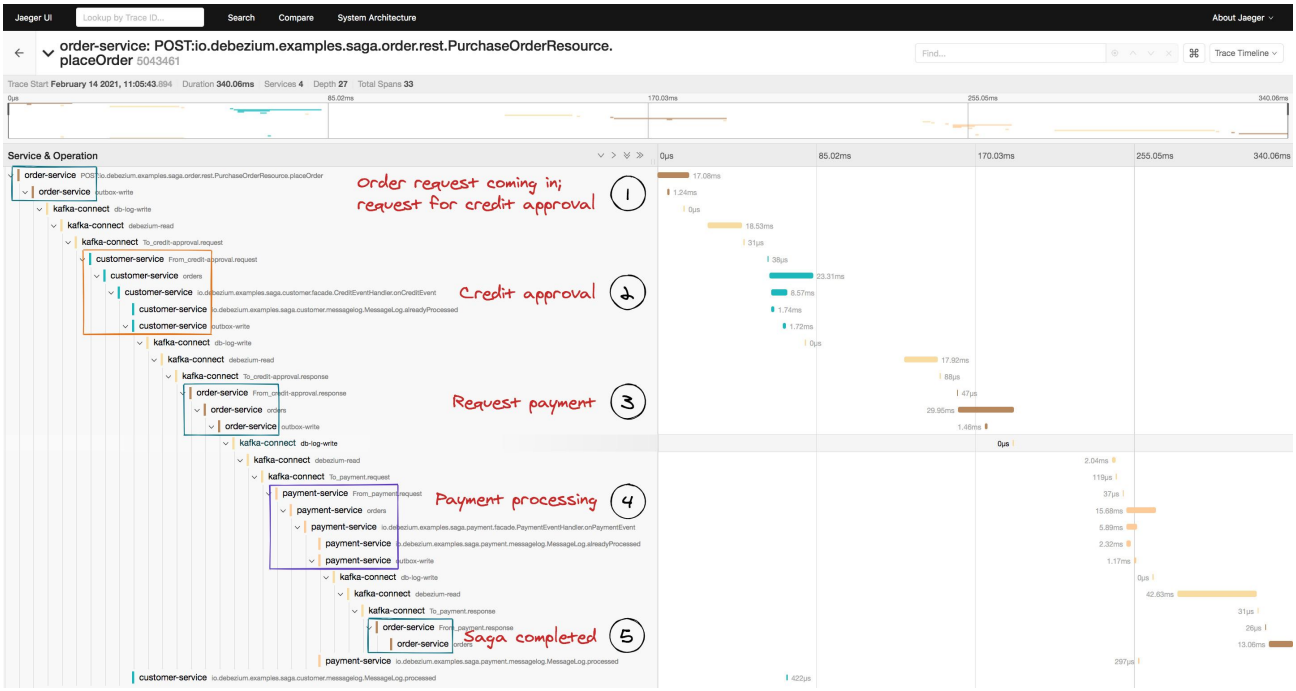


Figure 7. Saga flow in the Jaeger UI

The visualization in Jaeger nicely shows how the Saga flow is triggered by the incoming REST request in the order service (1), an outbox message is sent to customer (2) and back to order (3), followed by another one sent to payment (4) and finally back to order (5).

The tracing functionality makes it rather easy to identify unfinished flows—for example, because an event handler in one of the participating services fails to process a message—as well as performance bottlenecks, such as when one event handler takes unreasonably long to fulfill its part of the Saga flow.

Wrap-Up and Outlook

The Saga pattern offers a powerful and flexible solution for implementing long-running "business transactions," which require multiple, separate services to agree on either applying or aborting a set of data changes.

Thanks to the outbox pattern—implemented with CDC, Debezium and Apache Kafka—the Saga coordinator is decoupled from the availability of any of the other participating services. Temporary outages of single participants don't impact the overall Saga flow: once components come back up again, the Saga will continue from the point where it was interrupted before.

Of course, we should aspire for a service cut that reduces the need for interaction with remote services as much as possible. For instance, it might be an option to move the credit limit logic from the example to the order service itself, avoiding the coordination with the customer service. But depending on business requirements, the need for such interaction spanning multiple services might be impossible to avoid, in particular when it comes to integrating legacy systems, or systems that are not under our control.

When implementing complex patterns like Sagas, it's vital to exactly understand their constraints and semantics. Two things to be aware of in the context of the proposed solution are the inherent eventual consistency and the limited isolation level of the overarching business transaction. For instance, allocating a portion of the customer's credit limit could cause another order from that customer, that was submitted at the same time, to be rejected, also if the first order doesn't go through eventually.

The example project discussed in this article provides a PoC-level implementation for Saga orchestration based on CDC and the outbox pattern. It's organized into two parts:

- A generic "framework" component that provides the Saga orchestration logic in the form of a simple state machine along with the Saga execution log
- The specific implementation of the discussed order placement use case (the `OrderPlacementSaga` class shown in parts above, accompanying REST endpoints, etc.)

Going forward, we might extract the former part into a reusable component, for example, through the existing Debezium Quarkus extension. If there is interest in this, please let us know by reaching out on the Debezium mailing list. One potential feature to add would be the means of executing multiple Saga steps concurrently. Whether that's reasonable or not is a business decision, but supporting it wouldn't be hard from a technical perspective. Contention while updating the Saga state may become a critical issue in this case; the post Optimizations to scatter-gather sagas discusses potential solutions for this. It'd also be interesting to have a facility for monitoring and identifying Sagas that haven't been completed after some time.

The proposed implementation provides a means of reliably executing business transactions with “all or nothing” semantics across a span of multiple services. For use cases with more complex requirements, such as flows with conditional logic, you might take a look at existing workflow engines and business process automation tools, such as Kogito. Another interesting technology to keep an eye on is the MicroProfile specification for long-running activities (LRA), which currently is under development. The MicroProfile community also is discussing the integration with transactional outbox implementations like Debezium’s.

Many thanks to Hans-Peter Grahsl, Bob Roldan, Mark Little, and Thomas Betts for their extensive feedback while writing this article!

About the Author

Gunnar Morling is a software engineer and open-source enthusiast by heart. He is leading the Debezium project, a tool for change data capture (CDC). He is a Java Champion, the spec lead for Bean Validation 2.0 (JSR 380) and has founded multiple open source projects such as Layrry, Deptective and MapStruct. Prior to joining Red Hat, Gunnar worked on a wide range of Java EE projects in the logistics and retail industries. He's based in Hamburg, Germany. Twitter: @gunnarmorling

Discuss