Choreographed VS orchestrated sagas

In this lesson you will learn about the different ways sagas can be implemented or coordinated in the context of a microservices architecture.

Choreography-based Saga

When using choreography, the decision making on each step of the saga is performed by the saga participants themselves. No central coordinator is involved.

Here we have our Inventory and Identity microservices, and also a Trading microservice, whose job is only to receive the initial request for a purchase from the client.

The client places a purchase request for 5 potions. Trading then emits an event stating that a purchase has been requested. Remember that events are a type of message that are published so that any other services can subscribe to them and act accordingly. They are not sent directly to any service.

In this case, Inventory has subscribed to the event, and it proceeds to grant 5 potions to the user in its local DB. It then fires an items granted event, to which Identity is subscribed. Identity will go ahead and debit 25 gil from the user. Then it publishes a gil debited event.

Trading has subscribed to this event and, when it arrives, it knows that all operations must have completed successfully, so it replies to the client with a successful response.

But what if Identity is not able to debit the 25 gil from the user? In that case, a Gil debit failed event is raised by Identity, which Inventory is subscribed to. Inventory then executes its compensating transaction where it will take back the 5 potions originally granted to the user.

Trading was also listening for Gil debit failed events and, if it arrives, it knows that something went wrong, and it replies with failure to the client.

A key benefit of a choreography-based saga is how loosely coupled the participants are. Notice that neither Inventory or Identity have any knowledge of the other participants or about the fact that they are part of a bigger transaction. They just listen to an event, perform their operation, and fire another event. Since all the logic lives in each microservice, there also zero risk of getting any logic centralized anywhere.

But, there are also drawbacks. As you can tell from this diagram, it can be hard to build up a mental model of the whole process. In our example we only have a few participants, but what if we involved 5, 10 or more? Could you be able to tell what's going on by just looking at the diagram?

Also, it's unclear what's the current state of the saga. Have the items been granted? Has the gil been debited? Has something failed along the way? You would need to either ask each individual service or build some other view out of the published events to tell the current state.

Let's now see how this compares with an orchestration-based saga.

Orchestration-based Saga

In an orchestration-based saga, the coordination logic is centralized in an orchestrator service, which sends commands to each participant to tell them what to do.

In our example, the Trading microservice is now an orchestrator overseeing the overall transaction. Once again, the client sends the request to purchase 5 potions. Trading keeps track of this request in its local database and then sends a Grant items command to the Inventory microservice. Remember that commands are the type of message that are sent directly to a service as opposed to being published for anyone to subscribe to.

Inventory will go ahead and perform its local transaction where it grants 5 potions to the user. Then it publishes an Items granted event, which Trading is subscribed to.

Trading records the fact that items have been granted in its own DB and then sends a Debit gil command to Identity. Identity performs its local database update to debit 25 gil from the user and then publishes a Gil debited event.

Trading records this event in its local DB and then it knows the overall transaction has completed, at which point it can reply to the client with a successful response. Notice that the client does not need to necessarily sit and wait for a response from Trading. It could also just send the original request and then start polling Trading from time to time to ask if the transaction has completed.

Of course, there could be issues during the process. So, what happens if the gil cannot be debited in this case? Well then Identity fires a Gil debit failed event, which Trading is subscribed to. At that point, Trading keeps track of this failure and then sends a Subtract items command to Inventory, to which it responds by executing the compensating transaction that takes away the 5 potions originally granted to the user.

And then it replies with a failure to the client, if it was waiting for a response.

In terms of benefits, as you can see from this diagram, it is easier to understand the whole process, especially because Trading is the one sending commands to each service depending on the success or failure of the previous command. It's also easier to tell the current state of the overall process because Trading is keeping track of the result of each step.

There are also drawbacks. As you can imagine, we now have higher domain coupling because now our Trading microservice needs to know about each of the other participants in order to tell them what to do at each step. Also, there's the risk of centralizing too much in the orchestrator, which could result in a smart orchestrator but dumb microservices.

These issues can be mitigated to some extent by making sure that the only responsibility of the orchestrator is to drive the coordination or sequencing logic, but avoiding having any kind of business logic on them.

In this course you will implement an orchestration-based saga since I believe it's easier to reason about in a beginner level course like this one, and it should work well in most scenarios. However, feel free to opt for a choreography-based saga if you would like to go for a more decoupled approach and the additional complexity is not a concern for your team.

In the next module we will prepare our existing microservices to participate in the upcoming purchase saga.