**Orchestration-based sagas**

Orchestration is another way to implement sagas. When using orchestration, you define an orchestrator class whose sole responsibility is to tell the saga participants what to do. The saga orchestrator communicates with the participants using command/async reply-style interaction. To execute a saga step, it sends a command message to a participant telling it what operation to perform. After the saga participant has performed the operation, it sends a reply message to the orchestrator. The orchestrator then processes the message and determines which saga step to perform next.

***IMPLEMENTING THE CREATE ORDER SAGA USING ORCHESTRATION***

**Diagram

Description automatically generated**

The saga is orchestrated by the CreateOrderSaga class, which invokes the saga participants using asynchronous request/response. This class keeps track of the process and sends command messages to saga participants, such as Kitchen Service and Consumer Service. The CreateOrderSaga class reads reply messages from its reply channel and then determines the next step, if any, in the saga.

The Create Order Saga has four scenarios. In addition to the happy path, the saga can fail due to a failure in either Consumer Service, Kitchen Service, or Accounting Service. It’s useful, therefore, to model a saga as a state machine, because it describes all possible scenarios.

*MODELING SAGA ORCHESTRATORS AS STATE MACHINES*

A good way to model a saga orchestrator is as a state machine. A state machine consists of a set of states and a set of transitions between states that are triggered by events. Each transition can have an action, which for a saga is the invocation of a saga participant. The transitions between states are triggered by the completion of a local transaction performed by a saga participant. The current state and the specific outcome of the local transaction determine the state transition and what action, if any, to perform. There are also effective testing strategies for state machines. As a result, using a state machine model makes designing, implementing, and testing sagas easier.

This state machine consists of numerous states, including the following:

Text

Description automatically generated with medium confidence

Diagram

Description automatically generated

The state machine also defines numerous state transitions. For example, the state machine transitions from the Creating Ticket state to either the Authorizing Card or the Rejected Order state. It transitions to the Authorizing Card state when it receives a successful reply to the Create Ticket command. Alternatively, if Kitchen Service couldn’t create the Ticket, the state machine transitions to the Rejected Order state.

The state machine’s initial action is to send the VerifyConsumer command to Consumer Service. The response from Consumer Service triggers the next state transition. If the consumer was successfully verified, the saga creates the Ticket and transitions to the Creating Ticket state. But if the consumer verification failed, the saga rejects the Order and transitions to the Rejecting Order state. The state machine undergoes numerous other state transitions, driven by the responses from saga participants, until it reaches a final state of either Order Approved or Order Rejected.

**SAGA ORCHESTRATION AND TRANSACTIONAL MESSAGING**

Each step of an orchestration-based saga consists of a service updating a database and publishing a message. For example, Order Service persists an Order and a Create Order Saga orchestrator and sends a message to the first saga participant. A saga participant, such as Kitchen Service, handles a command message by updating its database and sending a reply message. Order Service processes the participant’s reply message by updating the state of the saga orchestrator and sending a command message to the next saga participant.

A service must use transactional messaging in order to atomically update the database and publish messages.

**BENEFITS AND DRAWBACKS OF ORCHESTRATION-BASED SAGAS**

**Text, letter

Description automatically generated**

Orchestration also has a drawback: the risk of centralizing too much business logic in the orchestrator. This results in a design where the smart orchestrator tells the dumb services what operations to do. Fortunately, you can avoid this problem by designing orchestrators that are solely responsible for sequencing and don’t contain any other business logic.

I recommend using orchestration for all but the simplest sagas. Implementing the coordination logic for your sagas is just one of the design problems you need to solve. Another, which is perhaps the biggest challenge that you’ll face when using sagas, is handling the lack of isolation. Let’s take a look at that problem and how to solve it in the next module.