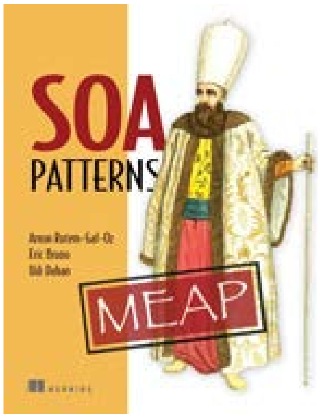
MANNING PUBLICATIONS



SOA Patterns
By Arnon Rotem-Gal-Oz, Eric Bruno, and Udi Dahan

Reservation pattern, which is closely related to the saga pattern, is a protocol level pattern that involves exchange of messages between service consumers and services. This article based on chapter 5 of SOA Patterns explains how to implement the reservation pattern.

To save 35% on your next purchase use Promotional Code **rotem0635** when you check out at http://www.manning.com/.

You may also be interested in…

Reservation Pattern

When you use transactions in *traditional* n-tier systems life is relatively simple. For instance, when you run a transaction and an error or fault occurs, you abort the transaction and easily roll back any changes, getting back your system-wide consistency and peace of mind. The reason this is possible is that a transaction isolates changes made within it from the rest of the world. One of the base assumptions behind transactions is that the time that elapses from the beginning of the transaction until it ends is short. Under that assumption we can afford the luxury of letting the transaction hold locks on our resources (such as databases) and prevent changes by others while the transaction is in progress. Transactions provide four basic guarantees: atomicity, consistency, isolation and durability, usually remembered by their acronym ACID.

Unfortunately, in a distributed world (SOA or otherwise), it is rarely a good idea to use atomic, short-lived transactions. Indeed, the fact that cross-service transactions are discouraged is one of the main reasons we'd consider using the saga pattern in the first place.

One of the obvious shortcomings of sagas is that you cannot perform rollbacks. Two conditions, locking and isolation, do not hold, so you cannot provide the needed guarantee. Still, since interactions, and especially long-running ones, can fail or be canceled, sagas offer the notion of compensations. Compensations are cool; we can't have rollbacks, so instead we will reverse the interaction's operation and have a pseudo-rollback. If we added one hundred units during the original activity, we'll just subtract the same 100 in the compensation. Easy, right? Wrong—as you probably know, it isn't easy.

Problem

Unfortunately, there are a number of problems with compensations. These problems come from the fact that, unlike ACID transactions, the changes made by the saga activities are not isolated. The lack of isolation means that other interactions with the service may operate on the data that was modified by an activity of other sagas and render the compensation impossible. To give an extreme example, if a request to one service changes the readiness status of the space shuttle to *all-set* and another service caused the shuttle to launch based on that status, it would be a little too late for the first service to try to reverse the *all-set* status now that the *bird has left the coop*. A more down-to-earth (pardon the pun) business scenario is any interaction where you work with limited resources, for example, ordering from a usually limited stock.

Consider the scenario in figure 1.

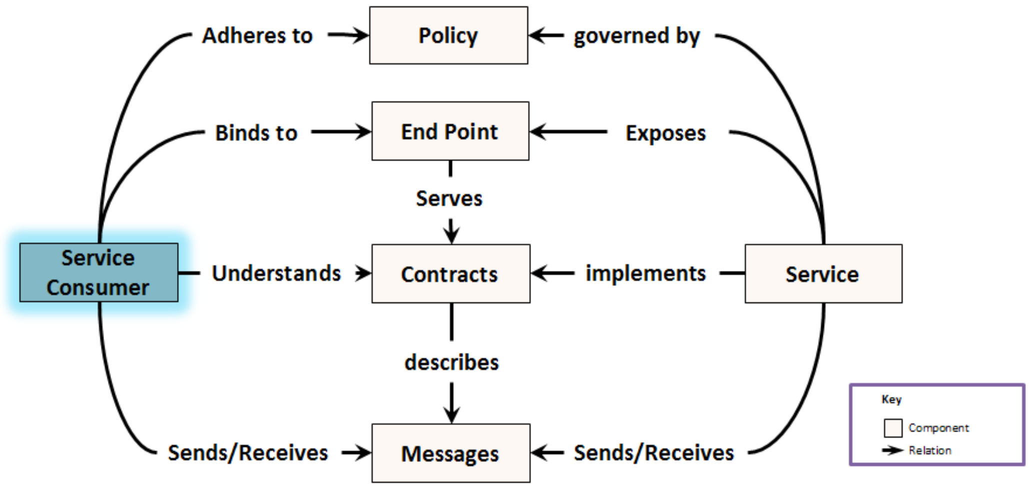


Figure 1 Connecting services with srvice consumers in the levels and layers beyond the basic message exchange patterns

A customer orders an item. The ordering service requests the item from the warehouse because it wants to ship the item to the customer (probably by notifying another service). Meanwhile, on the warehouse service, the item ordered causes a restocking threshold to be hit, triggering a restocking order from a supplier. Then the customer decides to cancel the order—now what?

Figure 2 Restocking order triggers a cancel order from a customer

Should the restocking order be cancelled as well? Can it be cancelled under the ordering terms of the supplier? Also a customer requesting the item between the ordering and cancellation might get an out of stock notice which will cause him to go to our competitors. This can be especially problematic for orders which are prone for cancellations like hotel bookings, vacations etc.

Another limitation of compensations and the saga pattern itself, for that matter, is that it requires a coordinator. A coordinator means placing trust in an external entity—outside (most) of the services involved in the saga—to set things straight. This is a challenge for some of the SOA goals because it compromises autonomy and introduces unwanted coupling to the external coordinator.

The question then is **how can we efficiently provide a level of guarantee in a loosely coupled manner while maintaining services' autonomy and consistency?**

We already discussed the limitations of compensations, which of course is one of the options to solve this challenge. Again, one problem is that we can't afford to make mini-changes since we will then be dependent on an external party to set the record straight. The other problem with compensations is that we expose these *semistates*, which essentially are the internal details of the services to the outside world. Increasing the footprint of the services' contract, especially with internal detail, makes the services less flexible and more coupled to their environment.

We've also mentioned that distributed transactions is not the answer since they both lock internal resources for too long (a saga might go on for days) as well as put excess trust on external services, which may be external to the organization. So what's the solution?

Solution

This seems like a quagmire of sorts; fortunately, real life already found a way to deal with a similar need for fuzzy, half guarantees—reservations! Implement the Reservation pattern and have the services provide a level of guarantee on internal resources for a limited time.

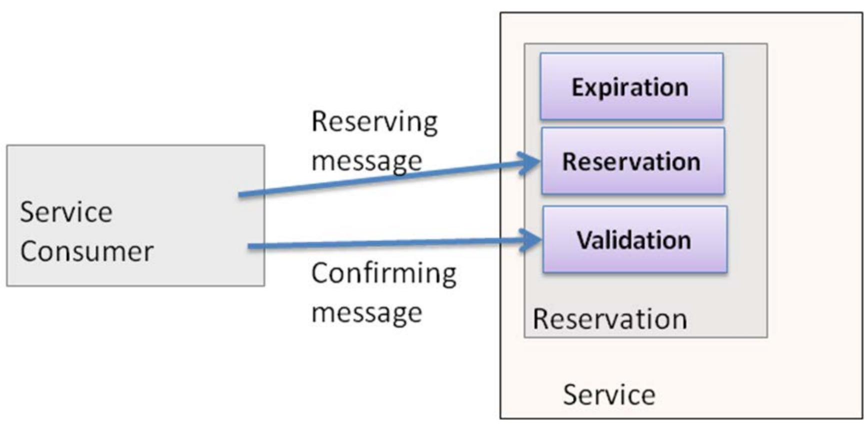


Figure 2 The reservation pattern. A service that implement reservation considers some messages as *reserving*, tries to secure an internal resource, and sends confirmation if it succeeds. When a message considered as *confirming* the service validates, the reservation still holds. In between, the service can choose to expire reservation based on internal criteria.

The Reservation pattern means there will be an internal component in the service that will handle the reservations. Its responsibilities include:

- *Reservation*—Making the reservation when a message that is deemed *reserving* arrives. For instance, when an order arrives, in addition to updating some durable storage (database) on the order, it needs to set a timer or an expiration time for the order confirmation; alternatively, it can set some marker that the order is not final.
- *Validation*—Making sure that a reservation is still valid before finalizing the process. In the ordering scenario mentioned before, that would be making sure the items designated for the order were not given to someone else.
- *Expiration*—Marking invalid reservation when the conditions changed. For example, if a VIP customer wants the item I reserved, the system can provision it for them. It should also invalidate my reservation, so when I finally try to claim it, the system will know it's gone. Expiration can also be timed, as in, "We're keeping the

book for you until noon tomorrow."

Reservations can be explicit—the contract would have a ReserveBook action or an implicit order. In case of an implicit order, the service decides internally what will be considered as a reserving message and what will be considered as a confirming message—an action like an order will trigger the internal reservation and an action like closing the saga will serve as the confirming message. When the reservation is implicit, the service consumer implementation will probably be simpler because the consumer designers are likely to treat reservation expiration as *simple* failures whereas when it is explicit they are likely to treat the reservation state.

Reservations happen in business transactions worldwide every day. The most obvious example is booking a hotel. You send in a request for a room (initiate a saga) saying you'd arrive on a certain date, say for a conference, and check out on another (complete the saga). The hotel says ok, we have a room for you (reservation), provided you confirm your arrival by a set date (limited time). Even if everything went well, you may still arrive at the hotel, only to find out your room has been given to another person (limited guarantee). The idea of the reservation pattern is to copy this behavior to the interaction of services so that services that support reservations offer a sort of *limited lock* for a limited time and with a limited level of guarantee. Limited level of guarantee means that, like real life, services can overbook and then resolve that overbooking by various strategies such as first come first served, VIP first served, and so on.

It is easy to see reservations applied to services that handle real-life reservations as part of their business logic, such as an ordering service for hotels (used in the example above) or an airline, and so on. However, reservations are suitable for a lot of other scenarios where services are called to provide guarantees on internal resources. For instance, in one system I built, we used reservations as part of the saga initiation process. The system uses the service instance pattern where some services are stateful (the reasons are beyond the scope of this discussion). Naturally, services have limited capacity to handle consumers (an instance can handle an n number of concurrent sagas/events). This means that, when a saga initialized, all the participants of the saga need to know the instances that are part of the saga. As long as a single service instance initiates sagas everything is fine. However, as illustrated in figure 3, when two or more services (or instances) initiate sagas concurrently they may (and given enough load/time they will) both try to allocate the same service instance to their relative sagas. In figure 3, we see that both initiator A and initiator B want to use participant A and participant B. Participant A has a capacity of 2, so everything is fine for both initiators. Service B, however, has limited capacity, so at least one of the sagas will have to fail the allocation and not start.

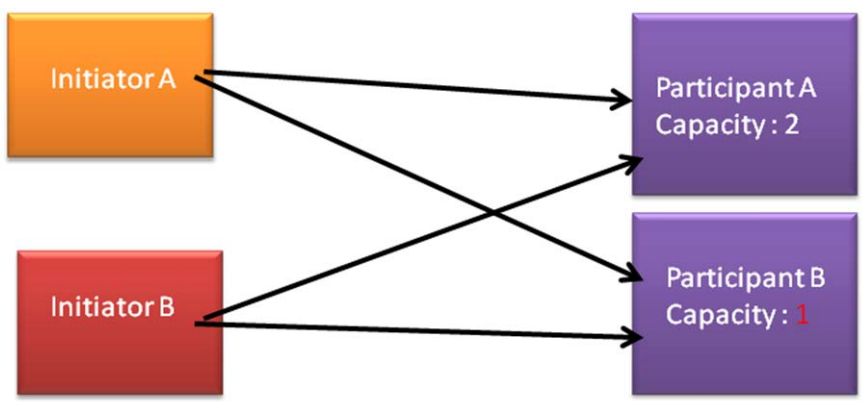


Figure 3 Sample for a situation that can benefit from the reservation pattern

The reservation pattern enabled us to manage this resource allocation process in an orderly manner by implementing a two-pass protocol (somewhat similar to a two phase commit). The initiator asks each potential participant to reserve itself for the saga. Each participant tries to reserve itself and notify back if it is successful. So, in the above scenario, A would say yes to both and B would say yes to one of them. If the initiator gets an OK from all of the involved services (within a timeout), it will tell all the participants the specific instances within the saga (initiate it).

The participants only reserve themselves for a short period of time. Once an internally set timeout elapses, the participants remove the commitment independently.

As a side note, I'll just say that the initiator and other saga members can't assume that the participant will be there just because they are officially part of the saga, and the system still needs to handle the various failure scenarios. The reservation pattern is used here only to help prevent overallocation and it does not provide any transactional guarantees.

A reservation is somewhat like a lock and it introduces some of the risks distributed locks present. These risks aren't inherent in the pattern but can easily surface if you don't pay attention during implementation (for example, using database locks for implementation):

- The first risk worth discussing is deadlock. Whenever you start reserving anything, especially in a distributed environment, you introduce the potential for deadlocks. For instance, if both participants had a capacity for single saga, initiator A contacts participant A first and participant B next, and initiator B used the reverse order, we would have had a deadlock potential. In this case, there are several mechanisms that prevent that deadlock. The first is inherent to the reservation pattern, where the participants release the lock themselves. However, for example, if there is a retry mechanism to initiate the sagas (because both would fail after the timeout) and the same resources will be allocated over and over, there may be a deadlock after all.
- Another risk to watch out for when implementing reservations is denial of service (DoS) whether maliciously or as a byproduct of misuse. Another way is via exploiting the reservations by constantly rereserving. Depending on the reservation timeout, regular firewalls might fail detecting the DoS so you may want to consider using a service firewall to help mitigate this thread.
- Besides the risks discussed above, another thing to pay attention to is when you introduce reservation because you are likely to add additional network calls. The system discussed above mentions that when it introduces another call to tell the saga members which instances are involved in the saga.

In addition to the service firewall pattern mentioned above, another pattern related to reservations is the active service pattern. The active service pattern can be used to handle reservation expiration when it is timed.

NOTE

Sometimes it's better, resource-wise, to handle expiration passively and not actively.

Technology mapping

The reservation pattern is more a business pattern than a technological one. This means there isn't a straight one-to-one technology mapping to make it happen. On the other hand, code-wise, the pattern is relatively easy to implement.

One thing you have to do is to keep a live thread at the service to make sure that when the lease or reservation expires someone will be there to clean up. One option is the active service pattern. You can use technologies supporting timed events that provide the wakeup service for you. For instance, if you are running in an EJB 3.0 server you can use single action timers whereby timers only raise their event once to accomplish this goal. The listing below shows a simple code excerpt to set a timer to go off based on the time received in a message. Other technologies provide similar mechanism to accomplish the same effect.

Listing 1 Setting a timer event (using JBOSS)

```
public class TimerMessage implements MessageListener {

  @Resource
  private MessageDrivenContext mdc;
    .
    .
    .
  public void onMessage(Message message) {
```

```
    ObjectMessage msg = null;
    try {                                                           #1
        if (message instanceof ObjectMessage) {
       msg = (ObjectMessage) message;
        TimerDetailsEntity e = (TimerDetailsEntity) msg.getObject();
        TimerService timerService = messageDrivenCtx.getTimerService();

       // Timer createTimer(Date expiration, Serializable info)    #2
       Timer timer = timerService.createTimer(e.Date, e);
       }
    } catch (JMSException e) {
       e.printStackTrace();
       mdc.setRollbackOnly();
    } catch (Throwable te) {
       te.printStackTrace();
    }
  }
.
.
.
```

<#1 Vanilla code to process a message and get the interesting entity out of it
<#2 Sets the single action timer based on the info in the message we've just got

Timer-based cancellation, as described above, might be overkill if the reservation implementation is simple. The reservation in the listing below, implemented in C#, is used by the participants.

Listing 2 Simple in-memory, nonpersistent reservation

```
public Guid Reserve(Guid sagaId)
       {
           try
           {
               Rwl.TryWLock();
               var isReserverd = Allocator.TryPinResource(localUri, sagaId);
               if (!isReserverd)                                     #1
                   return Guid.Empty;

              //Some code to set the expiration                    #2
               return sagaId;                                        #3

           }
           finally
           {

              Rwl.ExitWLock();
           }

   }
```

#1 The allocator is a resource allocation control that manages, among other things, the capacity of the service. If we didn't succeed in marking the service as belonging to the saga, we can't allocate the service to the specific saga.
#2 Here is where we need to add code to mark when the reservation expired. We'll try to do something other than timers.
#3 Successful reservation returns the SagaId. This assures the caller that the reply it got is related to the request it sent—a simple Boolean might be confusing.

Since the reservation in listing 2 does not involve heavy service resources (like, say, a database), we can implement a passive handling of reservation expiration, which will be more efficient than a timer-based one. The listing below shows a revised reservation implementation, which removes timeout reservation before it commits.

NOTE

An expired reservation can still be committed if no other reservation occurred in between or the capacity of the service is not exceeded.

Listing 3 Passive reservation expiration handling (added on top of the code from listing 2)

```
public Guid Reserve(Guid sagaId)
       {
           try
           {
```

```
            Rwl.TryWLock();
            RemoveExpiredReservations();                          #1
            var isReserverd = Allocator.TryPinResource(localUri, sagaId);
            if (!isReserverd)
                return Guid.Empty;

            OpenReservations[sagaId] = DateTimeOffset.Now + MAX_RESERVERVATION;   #2
            return sagaId;

        }
        finally
        {

           Rwl.ExitWLock();
        }

    }

 private void RemoveExpiredReservations()
    {
        var reftime = DateTimeOffset.Now;
        var ids = from item in OpenReservations where item.Value < reftime select
item.Key;
        if (ids.Count() == 0) return;
        var keys=ids.ToArray();
        foreach (var id in keys)
        {
            OpenReservations.Remove(id);
            Allocator.FreePinnedResources(id);
        }

    }
```

#1 Added a small method (RemoveExpiredReservations. which also appears in the listing) to clean expired reservations. This method is ran everytime the service needs to handle a new reservation request and it cleans up expired reservations. Note that there is no timer involved, reservation are only cleaned if there is a new reservation to process.
#2 Instead of a timer the reservation is done by marking down when the reservation will expire.

The code samples above show that implementing reservation can be simple. This doesn't mean that other implementations can't be more complex, for example, if you want/need to persist the reservation or distribute a reservation between multiple service instances but at its core it shouldn't be a heavy or complex process.

Another implementation aspect is whether reservations are explicit or implicit. Explicit reservation means there will be a distinct reserve message. This usually means there will also be a commit type message and that the service or workflow engine that requests the reservation might find itself implementing a two-phase commit type protocol, which isn't very pleasant, to say the least.

The other alternative is implicit whereby the service decides internally when to reserve, the conditions under which to commit the reservation and, when to reject it. As usual, the tradeoff is between simple implementation to the service and simple implementation for the service consumer.

Quality attributes

We wrap up the pattern by taking a brief look at some business drives (or scenarios) that can drive us to use the reservation pattern. In essence, the main drive to reservation is the need for commitment from resources and since it is a complementary pattern to Sagas it also has similar quality attributes. As mentioned above, reservation helps provide partial guarantees in long-running interactions; therefore the quality attribute that points us toward it is integrity.

Table 1 Reservation pattern quality attributes scenarios. These are the architectural scenarios that can make us think about using the Reservation pattern.

Quality attribute (level1)	Quality attribute (level2)	Sample scenario
Integrity	Correctness	Under all conditions, failure receive payment within 5 business days will

		cancel the order and shipping
Integrity	Predictability	Under normal conditions, the chances of a customer getting billed for a cancelled order shall be less than 5%

Reservation is a protocol level pattern that involves exchange of messages between service consumers and services.

Summary

Reservation deals with providing time-bound guarantees that allow consumers to work and coordinate with several services while avoiding distributed transactions. In this article, we showed you how to implement the reservation pattern.

Here are some other Manning titles you might be interested in:

Open Source SOA
Jeff Davis

SOA Governance in Action
Jos Dirksen

SOA Security
Ramarao Kanneganti and Prasad A. Chodavarapu

Last updated: January 2, 2012