Compensating Transaction pattern

06/23/2017 • 7 minutes to read • Contributors all

In this article

Undo the work performed by a series of steps, which together define an eventually consistent operation, if one or more of the steps fail. Operations that follow the eventual consistency model are commonly found in cloud-hosted applications that implement complex business processes and workflows.

Context and problem

Applications running in the cloud frequently modify data. This data might be spread across various data sources held in different geographic locations. To avoid contention and improve performance in a distributed environment, an application shouldn't try to provide strong transactional consistency. Rather, the application should implement eventual consistency. In this model, a typical business operation consists of a series of separate steps. While these steps are being performed, the overall view of the system state might be inconsistent, but when the operation has completed and all of the steps have been executed the system should become consistent again.

> The Data Consistency Primer provides information about why distributed transactions don't scale well, and the principles of the eventual consistency model.

A challenge in the eventual consistency model is how to handle a step that has failed. In this case it might be necessary to undo all of the work completed by the previous steps in the operation. However, the data can't simply be rolled back because other concurrent instances of the application might have changed it. Even in cases where the data hasn't been changed by a concurrent instance, undoing a step might not simply be a matter of restoring the original state. It might be necessary to apply various business-specific rules (see the travel website described in the Example section).

If an operation that implements eventual consistency spans several heterogeneous data stores, undoing the steps in the operation will require visiting each data store in turn. The work performed in every data store must be undone reliably to prevent the system from remaining inconsistent.

Not all data affected by an operation that implements eventual consistency might be held in a database. In a service oriented architecture (SOA) environment an operation could invoke an action in a service, and cause a change in the state held by that service. To undo the operation, this state change must also be undone. This can involve invoking the service again and performing another action that reverses the effects of the first.

Solution

The solution is to implement a compensating transaction. The steps in a compensating transaction must undo the effects of the steps in the original operation. A compensating transaction might not be able to simply replace the current state with the state the system was in at the start of the operation because this approach could overwrite changes made by other concurrent instances of an application. Instead, it must be an intelligent process that takes into account any work done by concurrent instances. This process will usually be application specific, driven by the nature of the work performed by the original operation.

A common approach is to use a workflow to implement an eventually consistent operation that requires compensation. As the original operation proceeds, the system records information about each step and how the work performed by that step can be undone. If the operation fails at any point, the workflow rewinds back through the steps it's completed and performs the work that reverses each step. Note that a compensating transaction might not have to undo the work in the exact reverse order of the original operation, and it might be possible to perform some of the undo steps in parallel.

> This approach is similar to the Sagas strategy discussed in Clemens Vasters' blog.

A compensating transaction is also an eventually consistent operation and it could also fail. The system should be able to resume the compensating transaction at the point of failure and continue. It might be necessary to repeat a step that's failed, so the steps in a compensating transaction should be defined as idempotent commands. For more information, see Idempotency Patterns on Jonathan Oliver's blog.

In some cases it might not be possible to recover from a step that has failed except through manual intervention. In these situations the system should raise an alert and provide as much information as possible about the reason for the failure.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

It might not be easy to determine when a step in an operation that implements eventual consistency has failed. A step might not fail immediately, but instead could block. It might be necessary to implement some form of time-out mechanism.

-Compensation logic isn't easily generalized. A compensating transaction is application specific. It relies on the application having sufficient information to be able to undo the effects of each step in a failed operation.

You should define the steps in a compensating transaction as idempotent commands. This enables the steps to be repeated if the compensating transaction itself fails.

The infrastructure that handles the steps in the original operation, and the compensating transaction, must be resilient. It must not lose the information required to compensate for a failing step, and it must be able to reliably monitor the progress of the compensation logic.

A compensating transaction doesn't necessarily return the data in the system to the state it was in at the start of the original operation. Instead, it compensates for the work performed by the steps that completed successfully before the operation failed.

The order of the steps in the compensating transaction doesn't necessarily have to be the exact opposite of the steps in the original operation. For example, one data store might be more sensitive to inconsistencies than another, and so the steps in the compensating transaction that undo the changes to this store should occur first.

Placing a short-term timeout-based lock on each resource that's required to complete an operation, and obtaining these resources in advance, can help increase the likelihood that the overall activity will succeed. The work should be performed only after all the resources have been acquired. All actions must be finalized before the locks expire.

Consider using retry logic that is more forgiving than usual to minimize failures that trigger a compensating transaction. If a step in an operation that implements eventual consistency fails, try handling the failure as a transient exception and repeat the step. Only stop the operation and initiate a compensating transaction if a step fails repeatedly or irrecoverably.

> Many of the challenges of implementing a compensating transaction are the same as those with implementing eventual consistency. See the section Considerations for Implementing Eventual Consistency in the [Data Consistency Primer](#) for more information.

When to use this pattern

Use this pattern only for operations that must be undone if they fail. If possible, design solutions to avoid the complexity of requiring compensating transactions.

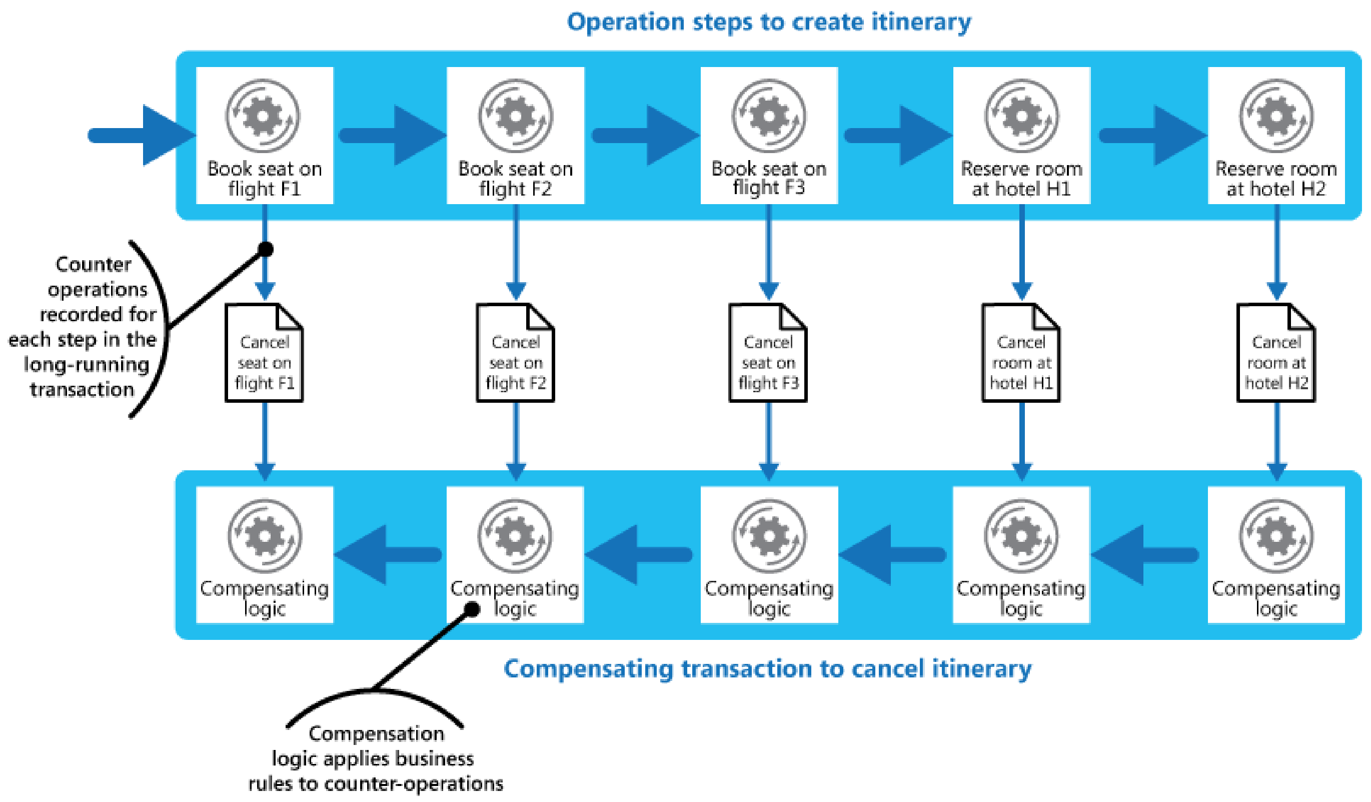
Example

A travel website lets customers book itineraries. A single itinerary might comprise a series of flights and hotels. A customer traveling from Seattle to London and then on to Paris could perform the following steps when creating an itinerary:

1. Book a seat on flight F1 from Seattle to London.
2. Book a seat on flight F2 from London to Paris.
3. Book a seat on flight F3 from Paris to Seattle.
4. Reserve a room at hotel H1 in London.
5. Reserve a room at hotel H2 in Paris.

These steps constitute an eventually consistent operation, although each step is a separate action. Therefore, as well as performing these steps, the system must also record the counter operations necessary to undo each step in case the customer decides to cancel the itinerary. The steps necessary to perform the counter operations can then run as a compensating transaction.

Notice that the steps in the compensating transaction might not be the exact opposite of the original steps, and the logic in each step in the compensating transaction must take into account any business-specific rules. For example, unbooking a seat on a flight might not entitle the customer to a complete refund of any money paid. The figure illustrates generating a compensating transaction to undo a long-running transaction to book a travel itinerary.



> ⓘ **Note**

It might be possible for the steps in the compensating transaction to be performed in parallel, depending on how you've designed the compensating logic for each step.

In many business solutions, failure of a single step doesn't always necessitate rolling the system back by using a compensating transaction. For example, if—after having booked flights F1, F2, and F3 in the travel website scenario—the customer is unable to reserve a room at hotel H1, it's preferable to offer the customer a room at a different hotel in the same city rather than canceling the flights. The customer can still decide to cancel (in which case the compensating transaction runs and undoes the bookings made on flights F1, F2, and F3), but this decision should be made by the customer rather than by the system.

Related patterns and guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- Data Consistency Primer. The Compensating Transaction pattern is often used to undo operations that implement the eventual consistency model. This primer provides information on the benefits and tradeoffs of eventual consistency.
- Scheduler-Agent-Supervisor pattern. Describes how to implement resilient systems that perform business operations that use distributed services and resources. Sometimes, it might be necessary to undo the work performed by an operation by using a compensating transaction.
- Retry pattern. Compensating transactions can be expensive to perform, and it might be possible to minimize their use by implementing an effective policy of retrying failing operations by following the Retry pattern.