Compensating Transaction pattern

Azure

When you use an eventually consistent operation that consists of a series of steps, the Compensating Transaction pattern can be useful. Specifically, if one or more of the steps fail, you can use the Compensating Transaction pattern to undo the work that the steps performed. Typically, you find operations that follow the eventual consistency model in cloud-hosted applications that implement complex business processes and workflows.

Context and problem

Applications that run in the cloud frequently modify data. This data is sometimes spread across various data sources 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 the eventual consistency model, a typical business operation consists of a series of separate steps. While the operation performs these steps, the overall view of the system state might be inconsistent. But when the operation finishes and all the steps have run, the system should become consistent again.

The Data Consistency Primer provides information about why distributed transactions don't scale well. This resource also lists principles of the eventual consistency model.

A challenge in the eventual consistency model is how to handle a step that fails. After a failure, you might need to undo all the work that previous steps in the operation completed. However, you can't always roll back the data, because other concurrent instances of the application might have changed it. Even in cases where concurrent instances haven't changed the data, undoing a step might be more complex than restoring the original state. It might be necessary to apply various business-specific rules. For an example, see the travel website that the Example section describes later in this article.

If an operation that implements eventual consistency spans several heterogeneous data stores, undoing the steps in the operation requires visiting each data store in turn. To

prevent the system from remaining inconsistent, you must reliably undo the work that you performed in every data store.

The data that's affected by an operation that implements eventual consistency isn't always held in a database. For example, consider a service-oriented architecture (SOA) environment. An SOA operation can invoke an action in a service and cause a change in the state that's held by that service. To undo the operation, you also have to undo this state change. This process 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 undo the effects of the steps in the original operation. An intuitive approach is to replace the current state with the state the system was in at the start of the operation. But a compensating transaction can't always take that approach, because it might overwrite changes that other concurrent instances of an application have made. Instead, a compensating transaction must be an intelligent process that takes into account any work that concurrent instances do. This process is usually application-specific, driven by the nature of the work that the original operation performs.

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, including how to undo the work that the step performs. If the operation fails at any point, the workflow rewinds back through the steps it has completed. At each step, the workflow performs the work that reverses that step.

Two important points are:

- A compensating transaction might not have to undo the work in the exact reverse order of the original operation.
- It might be possible to perform some of the undo steps in parallel.

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

A compensating transaction is an eventually consistent operation itself, so it can 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 fails, so you should define the steps in a compensating transaction as idempotent commands. For more information, see Idempotency Patterns on Jonathan Oliver's blog.

In some cases, manual intervention might be the only way to recover from a step that has failed. 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 you decide how to implement this pattern:

- It might not be easy to determine when a step in an operation that implements eventual consistency fails. A step might not fail immediately. Instead, it might get blocked. You might need to implement a time-out mechanism.
- It's not easy to generalize compensation logic. 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. If you do, the steps can be repeated if the compensating transaction itself fails.
- The infrastructure that handles the steps must meet the following criteria:
  - It's resilient in the original operation and in the compensating transaction.
  - It doesn't lose the information that's required to compensate for a failing step.
  - It reliably monitors the progress of the compensation logic.
- A compensating transaction doesn't necessarily return the system data to its state at the start of the original operation. Instead, the transaction compensates for the work that the operation completed successfully before it failed.
- The order of the steps in the compensating transaction isn't necessarily the exact opposite of the steps in the original operation. For example, one data store might be more sensitive to inconsistencies than another. The steps in the compensating transaction that undo the changes to this store should occur first.
- Certain measures can help increase the likelihood that the overall activity succeeds. Specifically, you can place a short-term, time-out–based lock on each resource

that's required to complete an operation. You can also obtain these resources in advance. Then, perform the work only after you've acquired all the resources. Finalize all actions before the locks expire.

- Retry logic that's more forgiving than usual can help 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 repeating the step. Stop the operation and initiate a compensating transaction only if a step fails repeatedly or can't be recovered.

- When you implement a compensating transaction, you face many of the same challenges that you face when you implement eventual consistency. For more information, see the "Considerations for Implementing Eventual Consistency" section in Data Consistency Primer.

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

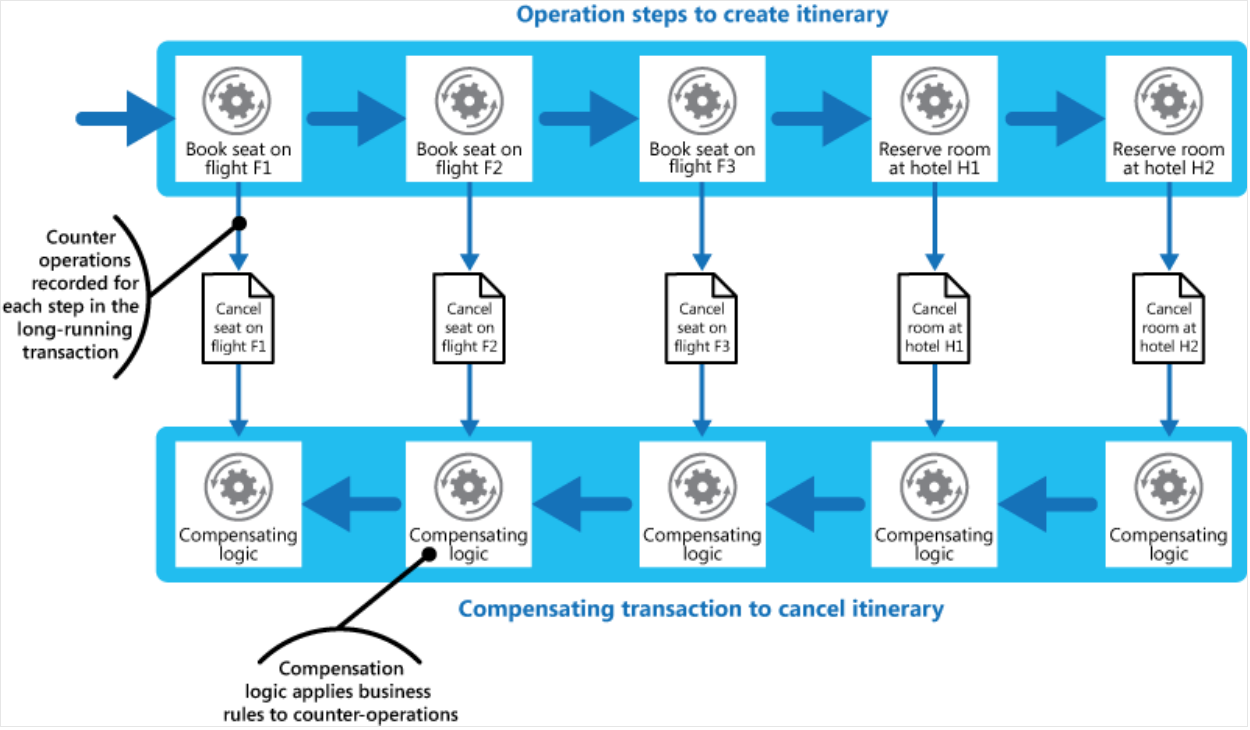
Customers use a travel website to book itineraries. A single itinerary might consist of a series of flights and hotels. A customer who travels from Seattle to London and then on to Paris might perform the following steps when creating an itinerary:

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. Besides performing these steps, the system must also record the counter operations for undoing each step. This information is needed in case the customer cancels the itinerary. The steps that are necessary to perform the counter operations can then run as a compensating transaction.

The steps in the compensating transaction might not be the exact opposite of the original steps. Also, the logic in each step in the compensating transaction must take business-specific rules into account. For example, canceling a flight reservation might not entitle the customer to a complete refund.

The following figure shows the steps in a long-running transaction for booking a travel itinerary. You can also see the compensating transaction steps that undo the transaction.



> **Note**
>
> You might be able to perform the steps in the compensating transaction in parallel, depending on how you design the compensating logic for each step.

In many business solutions, failure of a single step doesn't always necessitate rolling back the system by using a compensating transaction. For example, consider the travel website scenario. Suppose the customer books flights F1, F2, and F3 but can't 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 that case, the compensating transaction runs and undoes the bookings for flights F1, F2, and F3. But the customer should make this decision, not the system.

Next steps

- Data Consistency Primer. The Compensating Transaction pattern is often used to undo operations that implement the eventual consistency model. This primer provides information about the benefits and trade-offs of eventual consistency.
- Idempotency Patterns . In a compensating transaction, it's best to use idempotent commands. This blog post describes factors to consider when you implement idempotency.

Related resources

- Scheduler Agent Supervisor pattern. This article describes how to implement resilient systems that perform business operations that use distributed services and resources. In these systems, you sometimes need to use a compensating transaction to undo the work that an operation performs.
- Retry pattern. Compensating transactions can be computationally demanding. You can try to minimize their use by using the Retry pattern to implement an effective policy of retrying failed operations.
- Saga distributed transactions pattern. This article explains how to use the Saga pattern to manage data consistency across microservices in distributed transaction scenarios. The Saga pattern handles failure recovery with compensating transactions.
- Pipes and Filters pattern. This article describes the Pipes and Filters pattern, which you can use to decompose a complex processing task into a series of reusable elements. You can use the Pipes and Filters pattern with the Compensating Transaction pattern as an alternative to implementing distributed transactions.
- Design for self healing. This guide explains how to design self-healing applications. You can use compensating transactions as part of a self-healing approach.