Idempotency in microservices

In this lesson you will learn what idempotency is and how to add idempotency to your REST API operations to prevent unintended consequences.

The need for idempotency

To understand what is idempotency and why it is so important, let's first explore one unexpected scenario in our purchase process.

So, just as before, our client sends a request to purchase 5 potions to the Trading microservice. Trading publishes the purchase requested event and immediately tries to reply with an Accepted response.

However, this time, the response never reaches the client. This could happen for many reasons, like if the client or Trading loses its network connectivity temporarily, or maybe either the client or Trading just crashes in the middle of this request.

Let's imagine in this case that the client just lost its network connectivity temporarily. Trading is likely unaware of this, and since the purchase requested event was published, the purchase state machine is triggered and performs its usual steps of calculating the total gil and requesting Inventory to grant the items to the users inventory bag. Inventory grants the 5 potions, and the state machine moves to the Accepted state.

After the client recovers from the network connectivity issue, it's left wondering if Trading ever got the purchase request since it was not able to get a response. A logical thing to do then, for the client, is to retry the operation to honor the user's request.

The client sends the purchase request once again and Trading fires a purchase requested event just as before. This time the Accepted response from Trading successfully reaches the client. However, this has now triggered a new purchase process that will ultimately result in the user receiving 5 more potions in his inventory bag. As you can imagine, this is pretty bad.

But why did this happen? Well, to start with, the client went ahead and retried the operation without caring at all about the outcome of the original request. On the other side, the Trading microservice had no idea that this was a retry of a previous purchase request, since it received no hint about this in the request. Ultimately, Trading ends up duplicating the entire purchase.

So, what can we do here? It's time to talk about idempotency.

What is idempotency?

What is idempotency? Here for a simple definition I found on the web:

Idempotency is the ability to apply the same operation multiple times without changing the result beyond the first try.

Some operations are naturally idempotent, like when you send a PUT request to the Catalog Items REST API. No matter how many times you send a PUT request to update the antidote price to 8 gil, it will always result in antidotes to have a price of exactly 8 gil.

But other operations are not idempotent at all, like in our previous example. Each purchase request will result in a new purchase transaction, each of them granting items to the users inventory and debiting gil from his wallet.

Let's see how we can add idempotency to an operation that is not idempotent by nature, like in our purchase example.

Adding idempotency

Here is a revised version of our purchase example. The client once again sends the purchase request to the Trading microservice, but this time it sends one more thing along with it, an IdempotencyId. The idempotencyId is a unique value that both the client and Trading can use to refer to this specific purchase operation.

When Trading receives the request, it fires the purchase requested event including the IdempotencyId, so that the purchase state machine can also use it to keep track of the purchase. Just as before, the response never makes it back to the client, but regardless the purchase state machine starts and eventually asks Inventory to grant 5 potions to the users inventory bag. The state machine has moved to the Accepted state.

Again, the client recovers from the network issue and wonders if the purchase request was ever accepted. So, it decides to perform a retry, but this time it has been keeping track of the IdempotencyId and uses it as part of the retry request.

Trading gets the request and fires the purchase requested event with the IdempotencyId, after which it sends the Accepted response back to the client. Now, since Trading has also been keeping track of the IdempotencyId, it notices that it already has a state machine instance for that purchase, and that it is already in the Accepted state. So, there is no need to start a new purchase, and the duplication is avoided.

As you can see, a big benefit of using Idempotency here is that the client can now safely retry the purchase operation without ever worrying about duplicated transactions. No matter how many times the client retries, as long as it uses the same idempotencyId, the service will be able to easily correlate back to any previous operation and avoid starting a new purchase if the original one is already in progress or completed.

In the next lesson we will update our purchase requests to become idempotent.