Database Transactions

In this lesson you will learn about the need for transactions in your applications and how they work in the context of a monolithic system.

The need for transactions

To understand why you may need to use transactions in your applications, let's go back to our fictional game client where the player wants to purchase some items from the game store.

The player requests the purchase of some quantity of a specific item and then the application proceeds to grant such items to the player's inventory bag, but also it debits the corresponding amount of gil from the user's wallet.

What you have to think about here is what happens if the app fails to grant the items to the player even when it successfully debited the gil from the wallet? The player would certainly not be happy about spending gil to get nothing in return. But also, what if it is the other way around, and the problem is that the game was not able to debit gil from the player (because, perhaps the player did not have enough gil to spend on that number of items), and even so the items were granted to his inventory. That would result in a very happy player, but now our game economy is suffering an undesired inconsistency.

Here is where transactions come to the rescue. In a transaction a set of operations are treated as a single unit, in such a way that either all the operations complete successfully or all of them are rolled back together.

Therefore, with a transaction we can ensure that our game both grants items to the player and debits gil from him, or both these operations are undone as if none of them ever happened.

Transactions in a monolith

Let's see how transactions work in a monolithic system. Here our monolith includes only one service, the Trading service, which is responsible for performing the purchase operation by updating both the Inventory and Users tables in its associated relational database, or Inventory and Users collections in a NoSQL database like MongoDB.

Let's say the player requests the purchase of 5 potions, which the client app translates into a purchase request to the Trading service. The service then adds 5 potions on the Inventory table row for this player and then it reduces the amount of gil of the player by, say 25, in the Users table, assuming each potion costs 5 gil.

But, to make sure these both happen as a single operation, the service performs both actions in the context of a database transaction. This means the database will not consider any of the operations as completed until the service sends a commit command to the database. Only after this, any future query operations against any of the tables will find the updated values.

This type of transaction is usually known as an ACID transaction. ACID stands for:

- Atomicity: Since all operations must complete or all of them should fail
- Consistency: Because the database must be left in a valid, consistent state
- Isolation: Meaning that all operations can execute at the same time without interfering with each other
- And, Durability: because when the transaction completes, the data won't get lost even if there are unexpected system failures

Now, as you move to a microservices architecture, the main issue you are going to hit is trying to get atomicity across your operations, because each service now has its own database and, generally speaking, ACID transactions are meant to happen within a single database.

This is where you usually start thinking about distributed transactions, which is the topic of our next lesson.