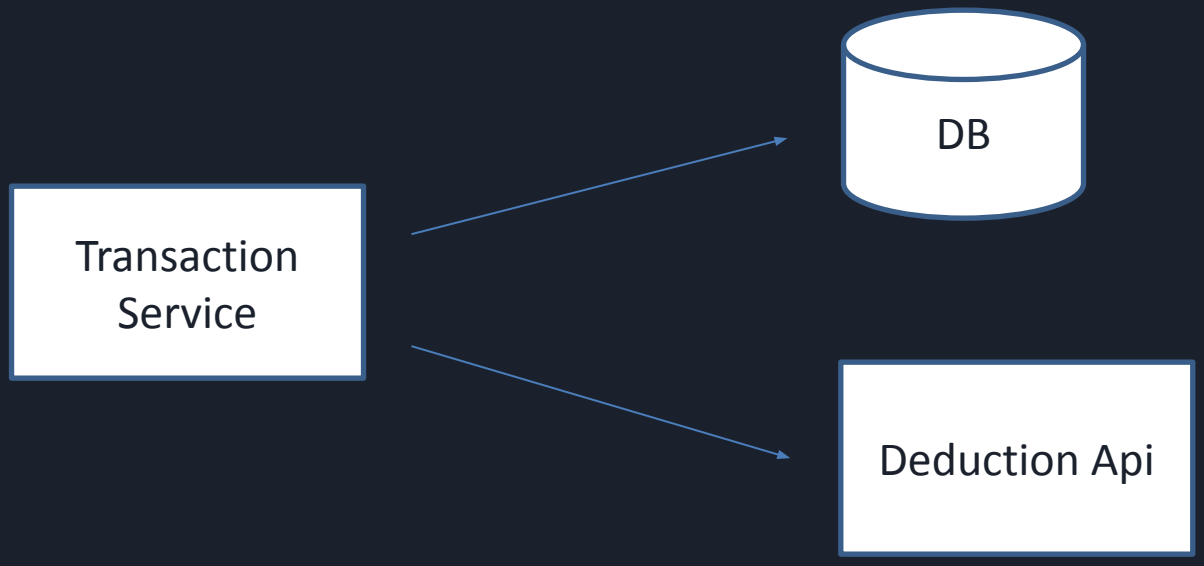
Handling Transactions in Distributed Systems

Agenda

- Context
- Objectives
- Example
- Design a Deduct API
  - Idempotence
- Design a Transaction process
  - without compensation
  - with compensation
- Extend
  - 2PC
- In conclusion

Context

1. There is a Transaction service and Deduction API
2. Service would call Deduct API, and insert a record into DB



Objectives

1. The total transaction amount should eventually match the user's balance
2. The system can automatically retry failed transactions, reducing manual intervention.

example 1

Step 1: Insert into DB $10

Step 2: Call Deduct API $10

-> **transaction: -10 balance: -10**

(Step 2 fail) Step 3: Delete record from Step 1

-> **transaction: 0 balance: 0**

(Step 3 fail)

-> **transaction: -10 balance: 0**

example 2

Step 1: Call Deduct API $10

Step 2: Insert into DB $10

-> **transaction: -10 balance: -10**

(Step 2 fail) Step 3: Call Compensate API $10

-> **transaction: 0 balance: 0**

(Step 3 fail)

-> **transaction: 0 balance: -10**

example 3: Transaction

Transaction

{

Step 1: Insert into Db $10

Step 2: Call Deduct API $10

Step 3: Transaction commit

(Step 2 fail): Transaction rollback

}

Risk of Transaction

1. DB lock (Shared locks, Exclusive locks)
2. If Deduct API return slowly, lock will be extended
3. The longer lock time, the more likely deadlock will occur
4. Performance overhead
5. Transaction time out

- **If is needed to use transactions, always ensure your transactions completed in shortest time**

Design a Deduct API

Idempotence

- An operation that can be applied multiple times without changing the result

(Idempotent)

Check Balance: /Wallet/GetBalance?userId=123

(Non-Idempotent)

Deduct: /Wallet/Deduct?userId=123&amount=10

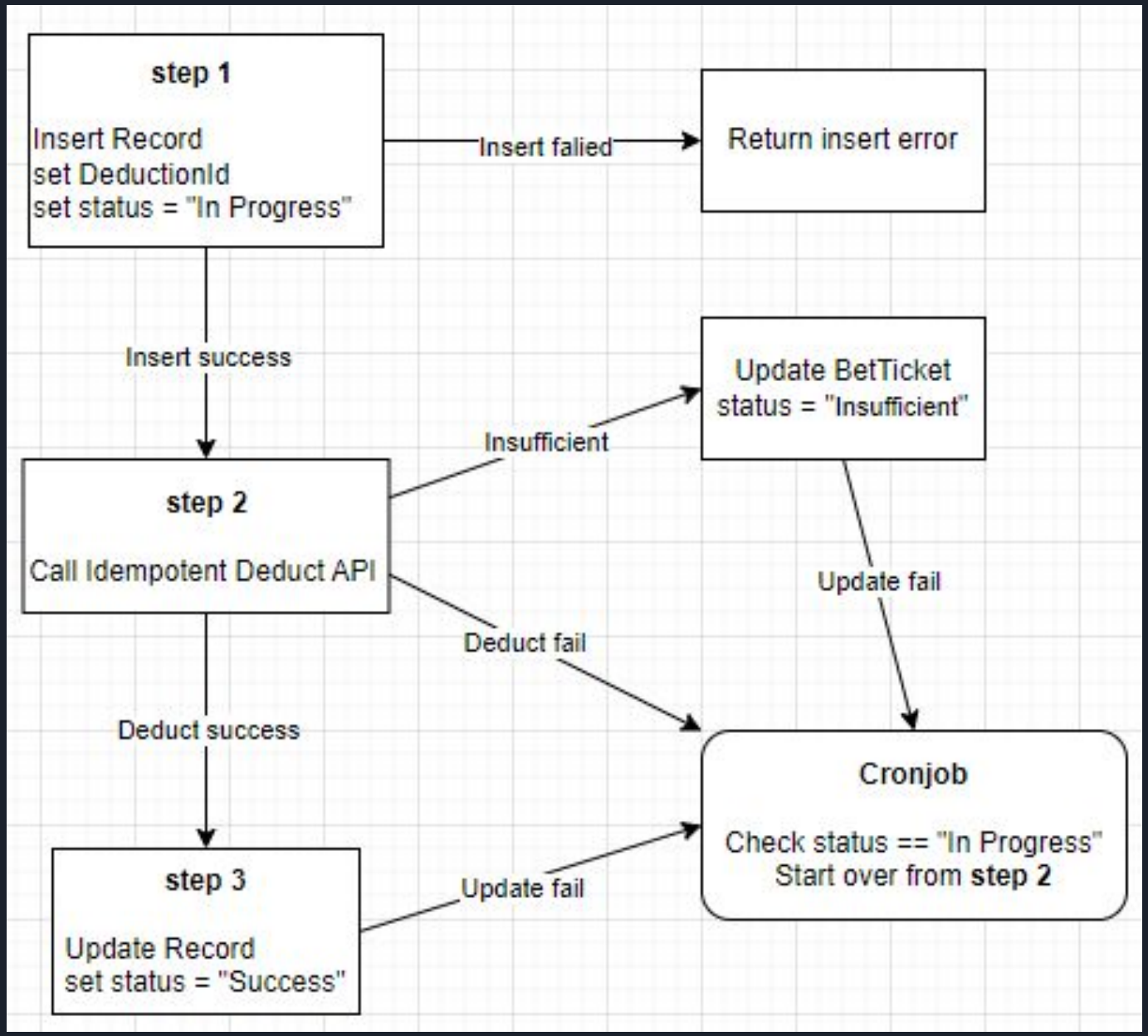
Design an Idempotent Deduction API

- Include a Idempotent Key in the request
- Before deducting, check if Idempotent key is exist
- If exist, don’t execute deduction
- example:

  /Wallet/Deduction?userId=123&amount=10&key=100001

Design Transaction process

Solution 1 (without compensation)


step 1
Insert Record
set DeductionId
set status = "In Progress"
Insert falied
Return insert error
Insert success
Update BetTicket
status = "Insufficient"
Insufficient
step 2
Call Idempotent Deduct API
Update fail
Deduct fail
Deduct success
Cronjob
Check status == "In Progress"
Start over from step 2
step 3
Update Record
set status = "Success"
Update fail

Solution 1 (without compensation)

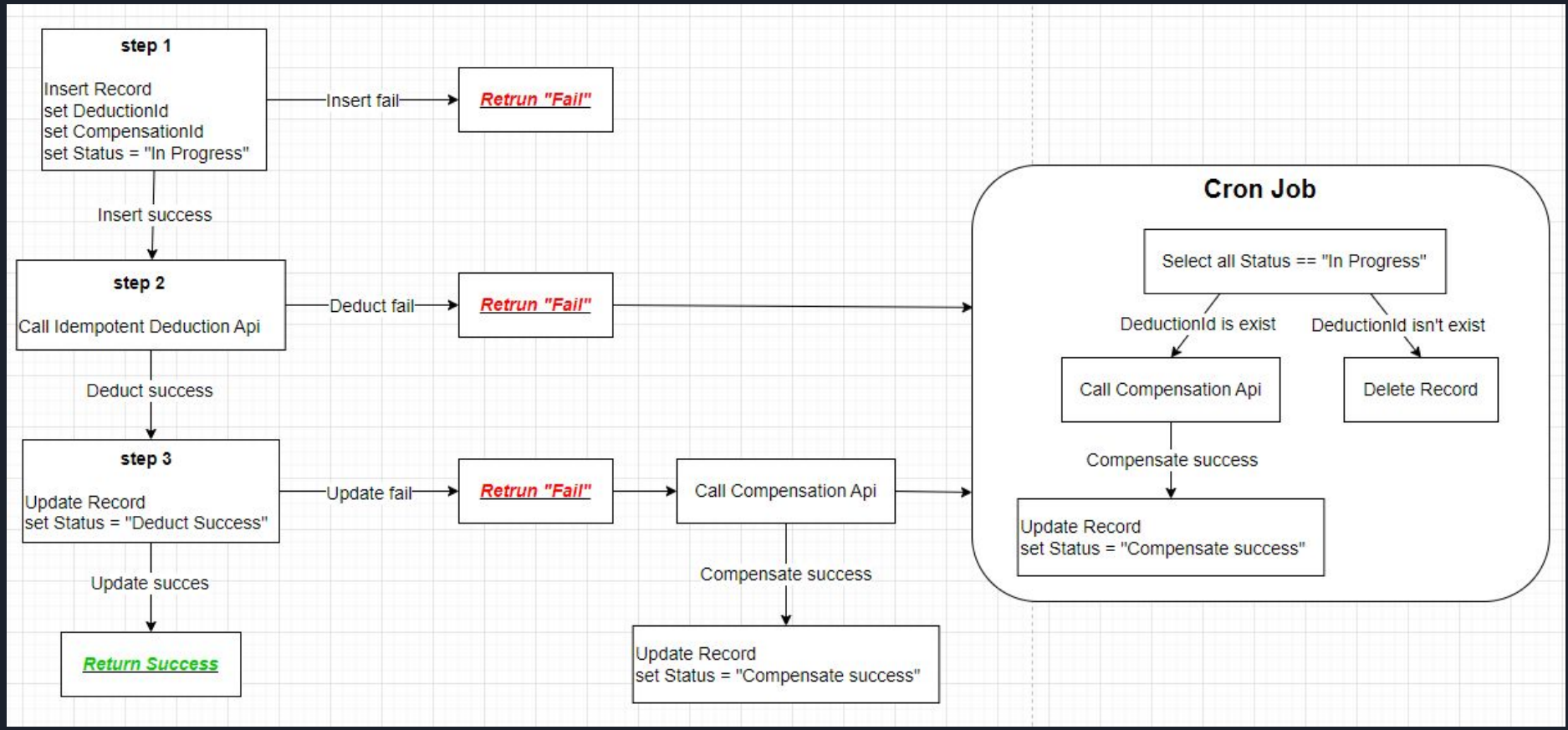
Step 1: Insert into DB with DeductionId, set Status = “In Progress”

Step 2: Call Idempotent Deduction API

Step 3: Update record, set Status = “Success”

- Cron Job will select all record which Status == “In Progress” and try to call Deduction API
- Use DeductionId as idempotent key to avoid duplicate deduction
- There will only 2 final Status, “Insufficient balance” and “Success”

Solution 2 (with compensation)


step 1
Insert Record
set DeductionId
set CompensationId
set Status = "In Progress"
Insert fail
Retrun "Fail"
Insert success
step 2
Call Idempotent Deduction Api
Deduct fail
Retrun "Fail"
Deduct success
step 3
Update Record
set Status = "Deduct Success"
Update fail
Retrun "Fail"
Call Compensation Api
Update succes
Return Success
Compensate success
Update Record
set Status = "Compensate success"
Cron Job
Select all Status == "In Progress"
DeductionId is exist
DeductionId isn't exist
Call Compensation Api
Delete Record
Compensate success
Update Record
set Status = "Compensate success"

Solution 2 (with compensation)

Step 1: Insert into DB with DeductionId, CompensationId, set Status = “In Progress”

Step 2: Call Idempotent Deduction API

Step 3: Update record, set Status = “Deduct Success”

- Return success only when all step is successful
- User would only see successful records
- Cron Job will select all record which Status == “In Progress” and try to call Compensation API
- DeductionId can also used to check if there is a deduction record

Solution 1 vs Solution 2

Solution 1 Pros and Cons

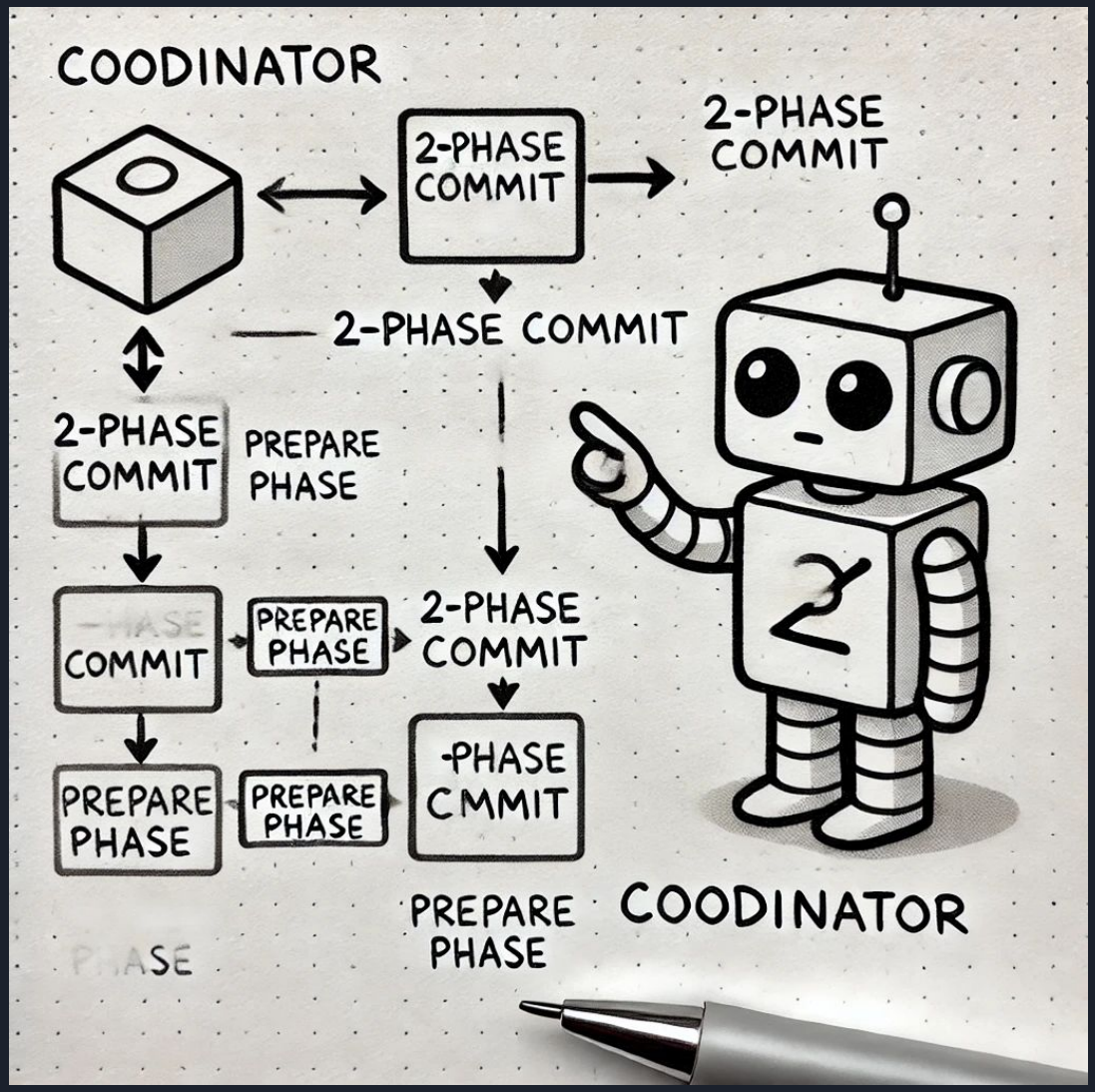
- Pros:
  - Without compensation, design is simpler
- Cons:
  - Worse user experience: If the record is used to show for user(like ticket in a game), the ticket may eventually fail

Solution 2 Pros and Cons

- Pros:
  - Better user experience: If the record is used to show for user(like ticket in a game), the ticket may not fail
- Cons:
  - With compensation, making process is more complex than Solution 1

Extended Topic

2 phase commit (2PC)



2 phase commit (2PC)

- There is a coordinator to handle all transaction process
- Phase 1 is prepare phase, coordinator will prepare all processes in a transaction
- Phase 2 is commit phase, coordinator will commit or abort the process
- If there any process is aborted, do compensation for all other processes

Why didn’t we use 2PC

- Have to maintain coordinator
- Increasing system complexity
- Still need to handle compensation flow if there any step is failed during commit phase

In Conclusion

- If is needed to use transactions, always ensure your transactions completed in shortest time
- Make Deduction API idempotent, include idempotent key in the request
- Always make your transaction process can safely redo no matter which step fails
- Simple design, case by case on your requirements

References

- https://chatgpt.com/
- https://en.wikipedia.org/wiki/Idempotence
- https://en.wikipedia.org/wiki/Two-phase_commit_protocol