Perché parlare di transazioni nelle API?

Molte operazioni reali sono multi-step:

- Conferma carrello
- Pagamento
- Creazione ordine
- Preparazione spedizione

Ogni step può essere una API separata con un proprio database.

Problema principale: Come garantire coerenza?

> Nelle API non esiste una "unica transazione ACID" che attraversa tutto.

Transazione locale vs Transazione distribuita

Transazione locale vs distribuita

Transazione locale

- Begin → Query → Commit / Rollback
- Un solo DB, un solo processo
- Atomica e veloce

Transazione distribuita

- Ogni servizio ha un proprio DB
- Comunicazione via HTTP/REST
- Fallimenti possibili in ogni step

> Le API non possono usare transazioni ACID cross-service.

❌ Perché le API NON possono usare transazioni ACID cross-service?

Le transazioni ACID funzionano **solo dentro un singolo database o un singolo sistema transazionale**.

Quando entriamo nel mondo delle API e dei microservizi, esistono:

- più processi indipendenti
- più database separati
- comunicazione via rete (HTTP)
- tempi diversi
- fallimenti indipendenti

E **ACID non è più applicabile**, ecco le 5 ragioni fondamentali.

---

1. ACID richiede un **transaction manager centralizzato**

In una transazione ACID classica:

- il transaction manager coordina tutte le operazioni
- tiene bloccate tabelle/righe finché non si esegue la COMMIT
- controlla che tutto sia atomico

Per le API non c'è nessun transaction manager centrale.

Ogni servizio:

- è un processo separato
- ha un DB separato
- non è coordinato da stessa istanza o runtime

---

2. ACID presuppone una **connessione affidabile e continua**

Una transazione ACID:

- apre una sessione DB
- esegue operazioni
- chiude con commit/rollback

In un sistema distribuito:

- la rete *non è affidabile*
- il servizio potrebbe essere down
- potrebbe non rispondere
- potrebbe rispondere in ritardo
- potrebbe rispondere due volte

Non è possibile mantenere aperta una transazione su HTTP tra servizi indipendenti.

---

3. ACID bloccherebbe i servizi (deadlock & lock distributed)

Se si provasse ad applicare ACID cross-service:

- Service A bloccherebbe righe/tabelle
- Service B bloccherebbe righe/tabelle
- tutti attendono il commit globale

Risultato: deadlock.

Esempio:

- Servizio Ordini blocca tabella ORDINI
- Servizio Pagamenti blocca tabella PAGAMENTI
- se uno dei due cade, tutto rimane bloccato

Nelle architetture moderne, i servizi devono rimanere **liberi**, non “bloccati” da transazioni remote.

---

4. I servizi sono autonomi (principio di *loose coupling*)

Microservizi = indipendenza:

- propria logica
- proprio DB
- proprie transazioni
- propria scalabilità

Una transazione ACID globale imporrebbe:

- schema condiviso
- commit coordinato
- accoppiamento rigido

Spezzerebbe l'architettura a microservizi.

---

5. ACID-cross-service diventa matematicamente fragile

Due servizi diversi:

- non scalano allo stesso modo
- non rispondono in contemporanea
- possono fallire indipendentemente
- possono generare timeouts asimmetrici

Il protocollo ACID *presuppone* sincronizzazione.
I servizi distribuiti *non possono sincronizzarsi* rigidamente.

Ecco perché 2PC (Two-Phase Commit) è teorico ma **quasi mai usato**.

Conclusione

> Le API REST nascono in un ambiente **stateless**, non transazionale e soggetto a fallimenti.
> Le transazioni ACID sono locali e non possono attraversare sistemi distribuiti.

Per questo motivo:

- ACID non può essere esteso a un sistema distribuito
- Le architetture moderne usano **Saga Pattern**, **eventi**, **compensazioni**
- I processi multi-step si modellano come **successions + rollback logico**, non come “un’unica magic transaction”.

Perché 2PC non funziona nel web moderno

Il Two-Phase Commit (2PC)

Coordinatore chiede:

- "Sei pronto a commit?"
- Tutti devono rispondere OK

Problemi:

- Lento
- Blocca risorse
- Fragile
- Poco adatto a sistemi distribuiti e non affidabili

> Non praticabile nei microservizi reali.

Introduzione al Saga Pattern

Il Saga Pattern

> Una saga è una sequenza di passi con possibili "operazioni compensative".

In una saga:

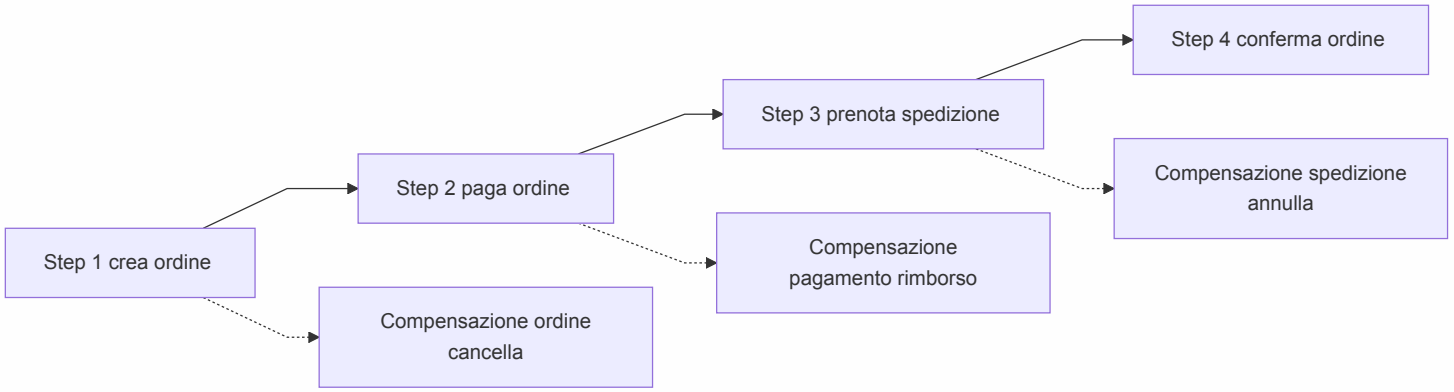
- Ogni step è una transazione locale
- In caso di errore → si eseguono compensazioni
- Il sistema converge a uno stato consistente

Esempio e-commerce

1. Crea ordine `PENDING`
2. Esegui pagamento
3. Prenota spedizione
4. Conferma ordine

Saga Pattern — Diagramma

Saga Pattern — Diagramma



Choreography vs Orchestration

Choreography vs Orchestration

Choreography

- Nessun coordinatore
- Servizi che reagiscono a eventi
- Decentralizzato

Orchestration

- Esiste un orchestratore
- Decide la sequenza
- Gestisce anomalie e compensazioni

> Libertà vs Controllo.

Il Saga Pattern è un modello.
Choreography e Orchestration sono due modi diversi per implementarlo.
Non si mescolano nella stessa Saga, ma possono coesistere nello stesso sistema.

Choreography e Orchestration nel Saga Pattern

Choreography e Orchestration **non convivono nella stessa saga**. Sono **due modalità alternative** per implementare una saga.

🕺 *Choreography*

- Nessun coordinatore centrale
- I servizi reagiscono a eventi
- Flusso distribuito
- Decisioni locali di ciascun servizio

🎼 *Orchestration*

- Esiste un orchestratore centrale
- Stabilisce la sequenza dei passi
- Coordina compensazioni e fallimenti
- Flusso esplicito e controllato

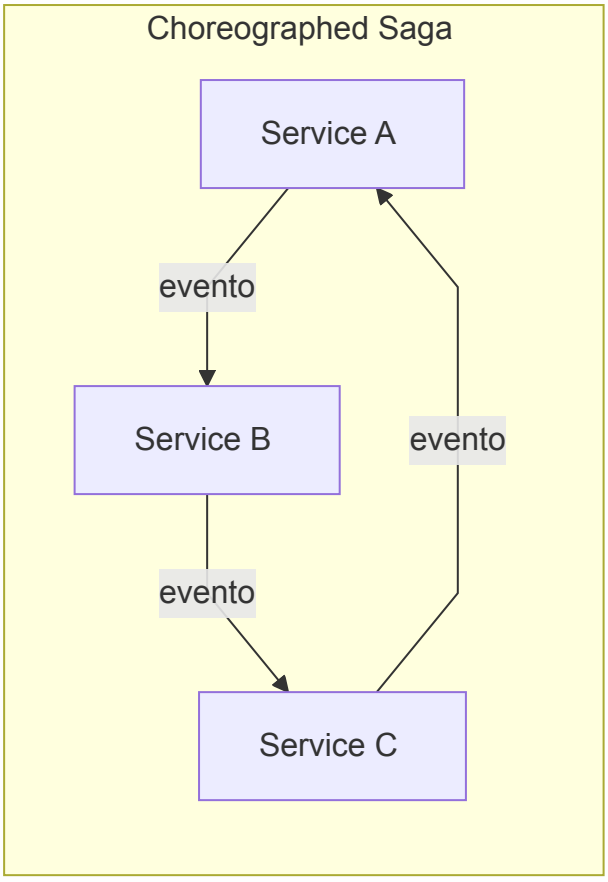
❗ *Possono convivere nello stesso sistema?*

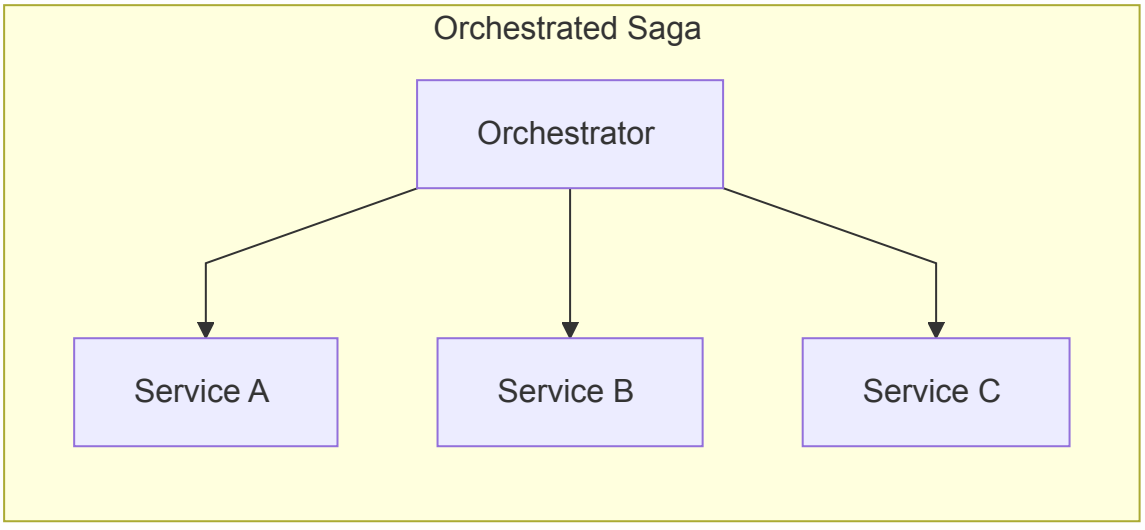
Sì. In un sistema complesso possono coesistere:

- saghe orchestrate (macro-processi)
- saghe coreografate (sotto-processi interni)

Ma ciascuna singola saga usa uno stile solo.

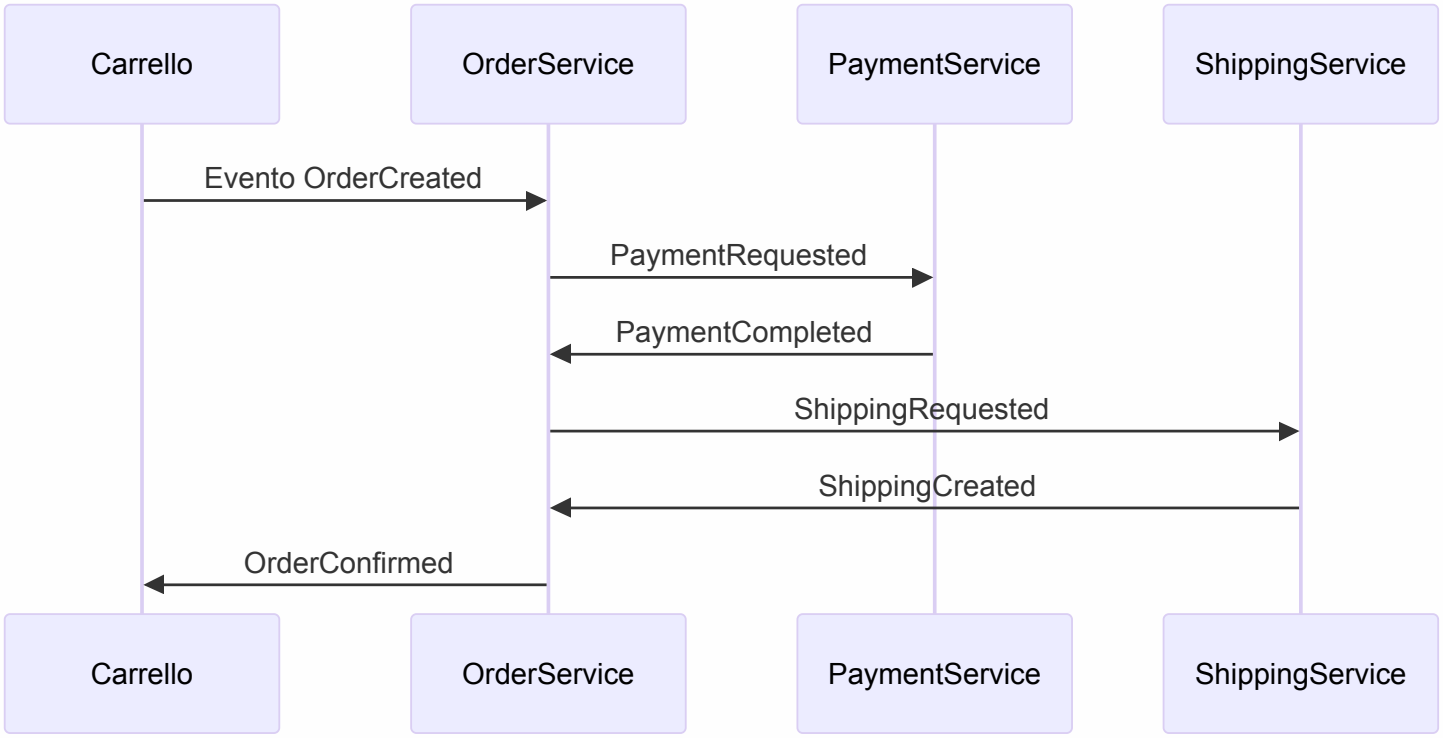
Diagramma: due modi distinti di implementare una Saga





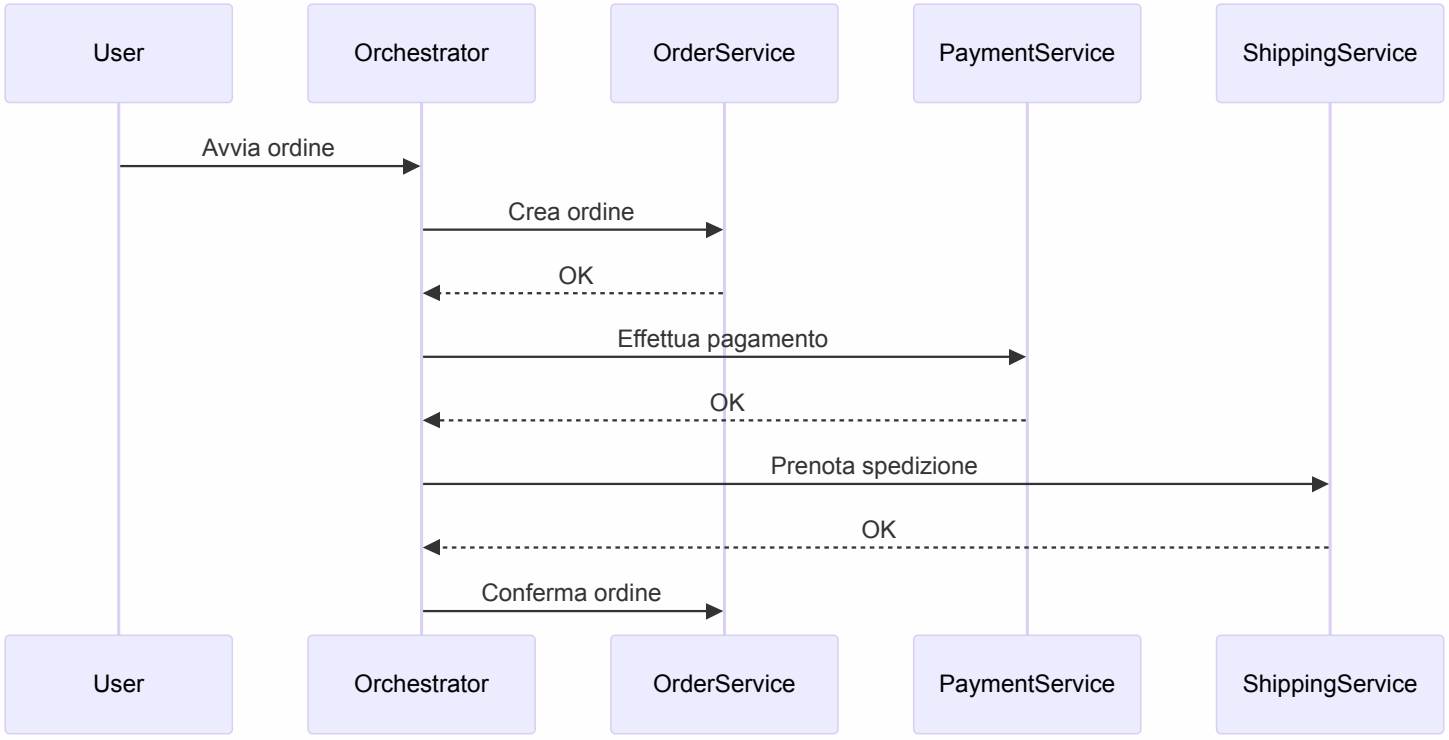
Saga tramite Eventi — Sequence Diagram

Saga — Choreography (Eventi)



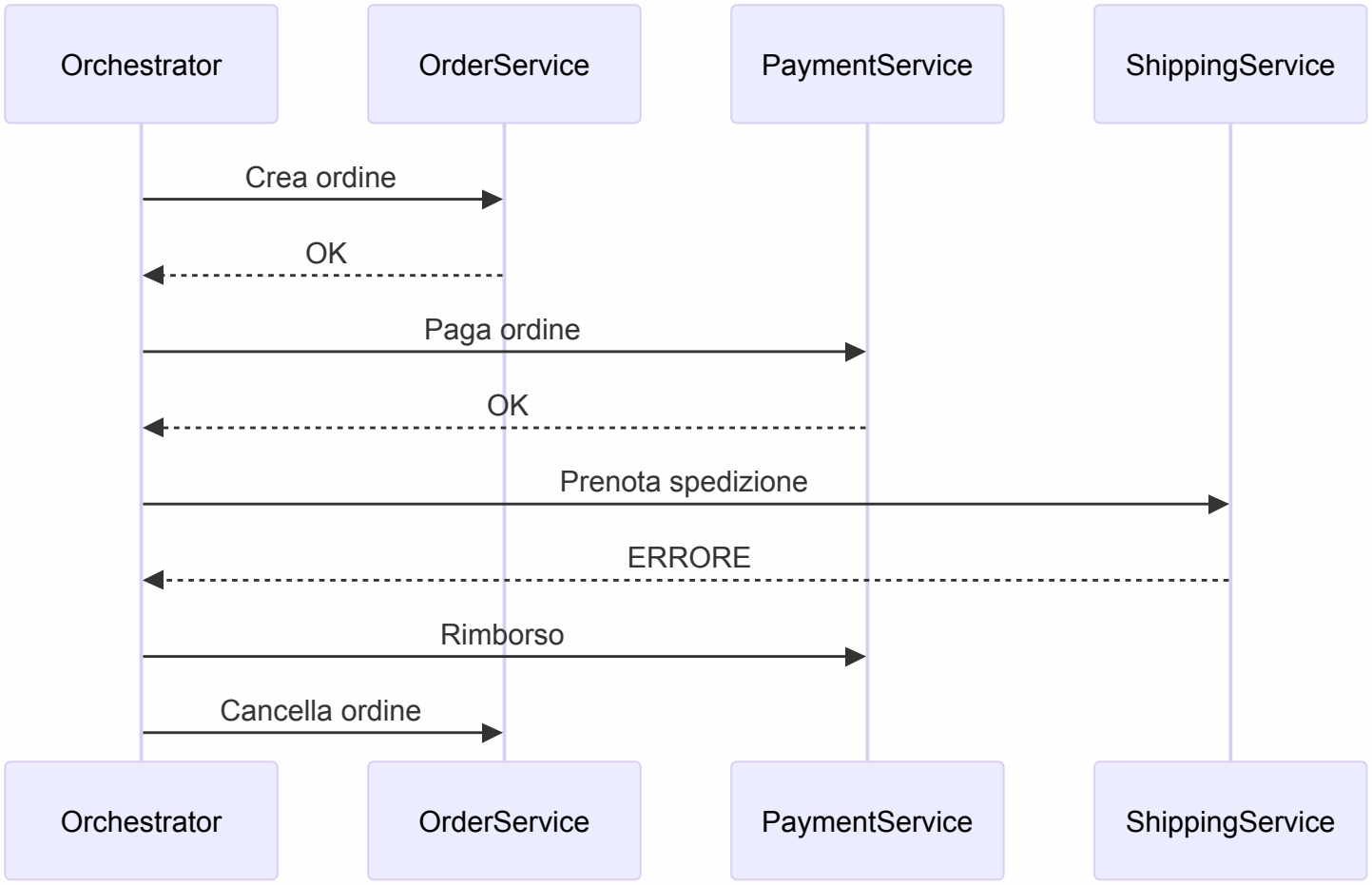
Saga con Orchestratore — Sequence Diagram

Saga — Orchestration



Saga con errore e compensazioni

Saga — Errore e compensazioni



Esempio di compensazioni in una Saga

Esempio di compensazioni in una Saga

Si consideri il processo di acquisto in un e-commerce:

1. Creazione ordine
2. Pagamento
3. Prenotazione spedizione
4. Conferma ordine

In una saga **ogni step è una transazione locale**. Se uno step fallisce, non esiste un rollback ACID globale: si esegue una **operazione di compensazione**.

Flusso di base

- Step 1: `OrderService` crea un ordine con stato `PENDING`
- Step 2: `PaymentService` esegue il pagamento
- Step 3: `ShippingService` prenota la spedizione
- Step 4: `OrderService` porta l'ordine in stato `CONFIRMED`

Compensazioni possibili

- Se **fallisce il pagamento**:
  - si annulla l'ordine (`CANCELLED`)
- Se **il pagamento va a buon fine ma fallisce la spedizione**:
  - si chiede un **rimborso** al `PaymentService`
  - si annulla l'ordine (`CANCELLED` oppure `ERROR` gestito)

Pseudocodice della logica di compensazione

```
crea_ordine()
risultato_pagamento = paga_ordine()

if risultato_pagamento == FALLITO:
    annulla_ordine()
else:
    risultato_spedizione = prenota_spedizione()
    if risultato_spedizione == FALLITO:
        rimborsa_pagamento()
        annulla_ordine()
    else:
        conferma_ordine()
```

Messaggio chiave

- La saga **non annulla tecnicamente** le operazioni gia committate.
- Applica invece operazioni di **compensazione** che riportano il sistema in uno stato coerente (ordine cancellato, cliente rimborsato, nessuna spedizione aperta).

> Il rollback non è magico: è una sequenza di azioni di business pensate in anticipo.

Stati robusti dell’ordine

Stati dell'ordine

Esempio ciclo di vita:

- PENDING
- PAID
- SHIPPING_REQUESTED
- SHIPPED
- CONFIRMED
- CANCELLED
- ERROR

> Stati robusti prevengono comportamenti inconsistenti.

Perché REST non basta

Perché REST non basta?

REST = request/response.

Transazioni distribuite richiedono:

- Eventi
- Code di messaggi
- Retry
- Stato persistente
- Orchestrazione o coreografia

> REST fa parte, ma non è tutto.

Idempotenza nelle saghe

Nelle saghe ogni step puo essere rieseguito piu volte a causa di retry, timeout o messaggi duplicati.

Obiettivo dell'idempotenza:
Ripetere la stessa operazione deve produrre lo stesso stato finale, senza duplicati e senza effetti collaterali doppi.

Esempi di idempotenza nei passi della saga

createOrder(orderId)

- Se l'ordine non esiste: crearlo
- Se esiste gia: restituire OK senza crearne uno nuovo

Esempio:

```
if ordine_esiste:
    return OK
else:
    crea_ordine
    return OK
```

payOrder(paymentId)

- Se il pagamento e gia stato processato: restituire il risultato precedente
- Altrimenti: processare il pagamento e salvarne lo stato

reserveShipping(shippingReqId)

- Se la prenotazione esiste: restituire la prenotazione
- Altrimenti: crearla

Idempotenza tecnica vs logica

Idempotenza tecnica: la stessa API puo essere richiamata piu volte senza errori tecnici.

Idempotenza logica: il significato di business resta corretto (nessun doppio pagamento, nessuna doppia spedizione, nessun duplicato).

Formula intuitiva

```
f(f(f(x))) = f(x)
```

> L'idempotenza non significa "non fare nulla". Significa che se la stessa richiesta arriva piu volte, il risultato finale deve essere coerente e sicuro.

Retry, Backoff e Jitter nelle saghe

Retry, Backoff e Jitter nelle saghe

Nelle architetture distribuite alcune chiamate possono fallire in modo temporaneo:

- rete lenta
- servizio momentaneamente non disponibile
- timeouts
- messaggi persi o duplicati

Per questo e fondamentale implementare **retry controllati**, evitando effetti collaterali o carico eccessivo.

Retry

Il retry consiste nel ripetere una chiamata che ha avuto un errore non definitivo.

Esempi di errori temporanei tipici:

- timeout
- HTTP 503 (Service Unavailable)
- rete congestionata

In una saga, ogni step deve essere progettato per essere idempotente per supportare i retry.

Backoff

Il backoff introduce un ritardo tra un tentativo e il successivo, cosi da:

- evitare sovraccarichi
- dare tempo al sistema di recuperare
- evitare 100 chiamate ripetute in pochi millisecondi

Tipologie comuni:

1. Fixed delay

Esempio: attendi 2 secondi ad ogni retry.

2. Exponential backoff

Ogni retry attende piu del precedente:

```
1s, 2s, 4s, 8s, 16s, ...
```

Permette di scaricare pressione su servizi in difficolta.

Jitter

Il jitter aggiunge **un ritardo casuale** ai retry per evitare che molti client ritentino nello stesso istante (thundering herd problem).

Esempio:

- backoff previsto: 4s
- jitter applicato: 4s +/- 30 percent

Questo crea un comportamento piu naturale e riduce i picchi di traffico.

Esempio integrato (semplificato)

```
retry_count = 0
backoff = 1s

while retry_count < max_retry:
    risultato = chiama_servizio()

    if risultato == OK:
        break

    attendi(backoff + jitter)
    backoff = backoff * 2
    retry_count = retry_count + 1
```

Collegamento con le saghe

- I retry sono essenziali quando si coordinano piu servizi.
- Ogni step della saga deve essere **idempotente** per poter essere ritentato.
- I retry devono essere limitati e con backoff, per evitare peggioramento del problema.
- Se dopo N retry un passo fallisce definitivamente, si attivano **le compensazioni** della saga.

> Retry controllati rendono la saga resiliente ai guasti temporanei senza generare disastri.

Thundering Herd Problem

Thundering Herd Problem

Il "Thundering Herd" e un fenomeno tipico nei sistemi distribuiti: molti client eseguono retry nello stesso identico momento.

Effetti:

- il servizio gia in difficolta riceve un picco di richieste
- peggiora ulteriormente la situazione
- aumento dei timeout
- comportamento a cascata dell'intero sistema

Perche accade?

Se i retry sono sincronizzati (esempio: tutti ritentano dopo 2 secondi), si genera un'onda di richieste simultanee.

Soluzione: Jitter

Aggiungere un ritardo casuale al backoff evita che i client ritentino esattamente nello stesso istante.

Mini diagramma (testuale)

```
Senza jitter:
Client A --- retry a t+2s
Client B --- retry a t+2s
Client C --- retry a t+2s  ---> overload

Con jitter:
Client A --- retry a t+2.1s
Client B --- retry a t+1.7s
Client C --- retry a t+2.4s  ---> carico distribuito
```

Retry in REST vs Retry in Event-Driven

Retry in REST vs Retry in Event-Driven

Retry in REST

- Il client ripete la chiamata HTTP
- Il server deve essere idempotente
- Il retry e sincrono (bloccante)
- Rischio di sovraccarico del servizio
- Necessita di backoff e jitter

Esempi tipici:

- HTTP 408 (timeout)
- HTTP 503 (service unavailable)

Retry in Event-Driven

- Il retry avviene tramite messaggi (queue/topic)
- Piu resiliente ai fallimenti temporanei
- Il consumatore puo riprovare automaticamente
- Non blocca il client
- Puo usare dead-letter-queue dopo N tentativi

Esempi tipici:

- Kafka
- RabbitMQ
- EventBridge

Conclusione

- REST retry: piu semplice, ma rischioso senza idempotenza rigorosa.
- Event-driven retry: piu robusto e naturale per saghe complesse.

Come implementare retry in Spring

Implementare Retry in Spring

Spring supporta nativamente meccanismi di retry controllato, utili nelle saghe.

1. Retry con Spring Retry (concetti)

- Possibilita di specificare numero di tentativi
- Possibilita di definire backoff (fisso o esponenziale)
- Possibilita di aggiungere jitter

Esempio concettuale:

```
@Retryable(maxAttempts = 3, backoff = 2s)
chiama_servizio_esterno()
```

2. Retry su chiamate REST

- utile quando un servizio remoto da timeout temporanei
- richiede idempotenza del metodo remoto

3. Retry su consumatori Event-Driven

- Spring Cloud Stream o Kafka Listener possono ritentare automaticamente
- se fallisce piu volte: messaggio trasferito in dead-letter-topic

4. Collegamento con le saghe

- ogni step della saga deve prevedere retry
- il retry deve essere sicuro (idempotente)
- il numero di retry deve essere limitato per attivare compensazioni

---

Retry locale vs Retry globale della saga

Retry locale vs Retry globale della saga

Retry locale (step-level)

- Applicato a un singolo passo della saga
- Eseguito dal servizio stesso o da una libreria (es. Spring Retry)
- Scopo: recuperare da errori temporanei
- Deve essere idempotente
- Esempi:
    - timeout del servizio Pagamenti
    - servizio Spedizioni momentaneamente non disponibile

Retry globale (saga-level)

- Gestito dall'orchestratore della saga o dal processo di business
- Decide se ripetere un passo, passare allo step successivo o attivare le compensazioni
- Entra in gioco quando i retry locali non bastano

Esempio:

- Dopo 3 retry locali di pagamento falliti, la saga considera il pagamento definitivamente fallito e attiva la compensazione dell'ordine (annullamento ordine).

Perche servono entrambi?

- Il retry locale aumenta la resilienza di ogni servizio rispetto a problemi temporanei.
- Il retry globale garantisce coerenza complessiva del processo di business.

Messaggio chiave:
Il retry locale prova a far funzionare il singolo step. Il retry globale decide quando smettere di riprovare e iniziare a compensare.

Cosa deve fare un API designer

Cosa deve saper fare un API designer

- modellare processi multi-step
- prevedere compensazioni
- definire stati robusti
- progettare API idempotenti
- documentare flussi success/failure
- considerare eventuali eventi e orchestrazione

> La complessità delle API distribuite è nella gestione del flusso, non nella sintassi.
