Zaroor. Yahan pesh hai woh mock interview ka code, jise aap sun kar bhi aasani se samajh sakte hain. Yeh aapke resume aur experience, khaas kar RBC Wealth Management project ke anubhav par aadharit hai.

**SCENE START**

**(Sound of a calm office room, a gentle hum)**

**Interviewer:** Namaste Dinesh, interview mein aapka swagat hai. Main aapka resume dekh raha tha. 8.5 saal ka anubhav, khaaskar Spring Boot aur Microservices mein, kaafi prabhavshali hai. Aap abhi Capgemini mein RBC ke Wealth Management project par kaam kar rahe hain, jiska architecture kaafi interesting lag raha hai.

**Dinesh (Candidate):** Namaste Sir. Ji, shukriya. Haan, RBC ka project kaafi challenging aur rewarding raha hai. Ismein humne ek legacy application ko poori tarah se modernize kiya hai, microservices architecture ka istemal karke.

**Interviewer:** Bahut accha. Toh chaliye, usi architecture se shuru karte hain. Aapke project mein kai microservices hain—Submission, Validation, Payment, etc. Yeh sab services aapas mein data share karti hain. Toh mera sawal yeh hai:

Aap distributed microservices mein transactional integrity aur eventual consistency kaise ensure karte hain?

**Dinesh (Candidate):** Ji sir, yeh microservices architecture ka ek bohot hi fundamental aur critical sawaal hai.

Dekhiye, ek traditional monolithic application mein, humare paas ACID transactions hote hain. Ek single database hota hai, aur hum begin transaction, commit, ya rollback karke data ki integrity maintain kar lete hain. Agar koi bhi step fail hua, toh poora transaction rollback ho jaata hai aur system consistent state mein rehta hai.

Lekin microservices mein, har service ka apna alag database ho sakta hai. Jaise humare project mein, SubmissionService ka apna data store hai, ValidationService ka apna, aur PaymentService ka apna. Yahan hum two-phase commit (2PC) jaisi cheezein use nahi kar sakte kyunki woh system ko tightly couple kar deta hai aur availability ko nuksaan pahunchata hai, jo microservices ke core principle ke khilaaf hai.

Is problem ko solve karne ke liye, humne **Saga Pattern** ka istemal kiya hai.

**Interviewer:** Theek hai. Saga Pattern. Kya aap ise apne RBC project ke sandarbh mein vistaar se samjha sakte hain?

**Dinesh (Candidate):** Bilkul, Sir. Ek Saga, asal mein, ek sequence of local transactions hai. Har transaction ek alag microservice ke andar hota hai. Jab ek service apna kaam poora kar leti hai, toh woh ek event publish karti hai. Yeh event agli service ko trigger karta hai ki woh apna kaam shuru kare.

Ab aate hain aapke sawaal ke doosre hisse par – integrity kaise maintain karein agar beech mein kuch fail ho jaaye?

Yahan par **Compensating Transactions** ka concept aata hai. Har local transaction ke liye, hum ek corresponding compensating transaction bhi design karte hain. Compensating transaction ka kaam hota hai pehle wale transaction ke asar ko undo karna.

Chaliye, main aapko humare project ke flow se samjhata hoon:

1. **Step 1: Submission**  
   Ek insurer request submit karta hai. SubmissionService is request ko receive karti hai, apne database mein ek ticket banati hai (yeh uska local transaction hai), aur phir ek event publish karti hai, jaise INSURANCE\_REQUEST\_SUBMITTED, Apache Kafka topic par.
2. **Step 2: Validation**  
   ValidationService is event ko sunti hai. Woh ticket ko validate karti hai. Agar validation kamyaab raha (local transaction successful), toh woh ek naya event publish karti hai: REQUEST\_VALIDATED.
3. **Step 3: Payment**  
   Ab PaymentService is REQUEST\_VALIDATED event ko consume karti hai aur payment process karne ki koshish karti hai.

Ab, yahan par do scenario ho sakte hain:

* **Happy Path:** Payment successful ho jaata hai. PaymentService PAYMENT\_PROCESSED event publish karti hai, aur Saga aage badh jaata hai.
* **Failure Path:** Maan lijiye, payment fail ho gaya. Payment gateway se error aa gaya. Ab system inconsistent state mein ja sakta hai—kyunki ticket submit ho chuka hai, validate bhi ho chuka hai, lekin payment nahi hua.

Isi ko handle karne ke liye hum **compensating transactions** ka istemal karte hain:

* PaymentService ek failure event publish karegi, jaise PAYMENT\_FAILED.
* Ab ValidationService, jo is failure event ko sun rahi hai, apna compensating transaction chalayegi. Uska kaam hoga ticket ka status 'VALIDATED' se badal kar 'VALIDATION\_FAILED' ya 'CANCELLED' karna. Phir woh ek VALIDATION\_REVERTED event publish karegi.
* SubmissionService is VALIDATION\_REVERTED event ko sunkar apne ticket ka status 'SUBMITTED' se 'FAILED' kar degi.

Is tarah, poora business process eventually ek consistent state mein aa jaata hai, bhale hi woh failure state ho. Data integrity bani rehti hai.

**Interviewer:** Interesting. Toh aapne Saga implement karne ke liye kaunsi approach use ki hai – Choreography ya Orchestration?

**Dinesh (Candidate):** Sir, humne ek mixed approach use ki hai, lekin primarily humara design **Choreography-based Saga** par nirbhar hai.

**Choreography** mein, har service event publish karti hai aur doosri services un events ko sunkar react karti hain. Koi central controller nahi hota. Isse services loosely coupled rehti hain. Jaise maine abhi example diya, woh choreography ka hi tha.

Iska fayda yeh hai ki system bohot resilient aur simple to design hota hai. Lekin iska ek nuksaan bhi hai – jab services badh jaati hain, toh workflow ko track karna mushkil ho jaata hai.

Wahin doosri taraf **Orchestration** mein ek central orchestrator service hoti hai, jo sabhi steps ko control karti hai. Woh SubmissionService ko call karegi, phir ValidationService ko, aur phir PaymentService ko. Agar kahin failure hota hai, toh orchestrator ki hi zimmedari hoti hai compensating transactions ko trigger karne ki.

Humne simpler workflows ke liye choreography use ki, aur jahan complex branching logic ya 10-12 services involved the, wahan humne ek lightweight orchestrator (jo khud ek Spring Boot service hai) banaya.

**Interviewer:** Yeh to theek hai. Lekin is event-driven approach mein ek aur problem hai. Kya ho agar aapki service apne database mein transaction commit kar de, lekin uske turant baad crash ho jaaye, Kafka par event publish karne se pehle? Tab toh system hamesha ke liye inconsistent ho jaayega. Ise kaise handle kiya?

**Dinesh (Candidate):** Excellent question, Sir. Yeh ek bohot hi practical aur common failure scenario hai. Is problem ko solve karne ke liye humne **Transactional Outbox Pattern** ka istemal kiya hai.

Is pattern mein, hum event ko direct message broker yaani Kafka par nahi bhejte.

Balki, jab SubmissionService apna local database transaction karti hai (ticket create karti hai), usi transaction ke andar, hum ek OUTBOX naam ke table mein event ka data bhi save kar lete hain.

Kyunki yeh dono operations—ticket create karna aur outbox table mein entry karna—ek hi database transaction ka hissa hain, yeh atomic hote hain. Ya toh dono honge, ya dono nahi honge.

Ab, ek alag, asynchronous process (jaise Debezium, ya ek simple scheduler) is OUTBOX table ko continuously monitor karta hai. Jaise hi use koi nayi entry dikhti hai, woh us event ko uthakar reliably Kafka mein publish kar deta hai aur phir us entry ko outbox table se delete ya 'processed' mark kar deta hai.

Isse yeh guarantee ho jaata hai ki event sirf tabhi publish hoga jab local transaction successfully commit ho chuka ho. Agar service transaction commit karne ke baad aur event publish karne se pehle crash ho bhi jaaye, toh restart hone par, outbox poller us pending event ko uthakar bhej dega. Isse "at-least-once" delivery ki guarantee milti hai.

**Interviewer:** Bilkul sahi. Aur is poore process se hi "Eventual Consistency" ka concept saamne aata hai, hai na?

**Dinesh (Candidate):** Ji, Sir, bilkul. "Eventual Consistency" ka matlab hi yahi hai ki system turant consistent state mein nahi hoga, lekin thode samay ke baad (eventually), sabhi services synchronize ho jaayengi aur data consistent ho jaayega.

Humare business use-case mein, kuch milliseconds ya seconds ka delay acceptable tha. Insurer ko turant usi microsecond mein final confirmation nahi chahiye hota. Isliye, Saga pattern aur eventual consistency humare liye perfect architectural trade-off tha. Yeh humein scalability, resilience, aur loose coupling deta hai, jo humare modern cloud-native application ke liye zaroori tha.

**Interviewer:** Bahut badhiya, Dinesh. Aapne Saga, Compensating Transactions, Choreography vs Orchestration, aur Transactional Outbox Pattern ko apne live project ke example ke saath bohot hi gehraai aur spashtata se samjhaya. Saaf dikhta hai ki aapne is par hands-on kaam kiya hai.

**Dinesh (Candidate):** Shukriya, Sir.

**(Sound of a pen clicking, gentle satisfaction)**

**SCENE END**